ITANIUM® SOLUTIONS
ALLIANCE

SUCCESS IS YOURS FOR THE TAKING. JOIN US.

Intel Itanium® Architecture Overview

CISC – RISC – EPIC

Terms in Time

CISC (Complex Instruction Set Computing-pre 1984)

RISC (Reduced Instruction Set Computing – post 1985)

- Goal: to optimize performance with simpler instructions (this effort coined the term CISC)

EPIC (Explicitly Parallel Instruction Computing – start 1986)

- Goal: to move beyond RISC performance bounds with explicitly-parallel instruction streams
- Implementation 1995

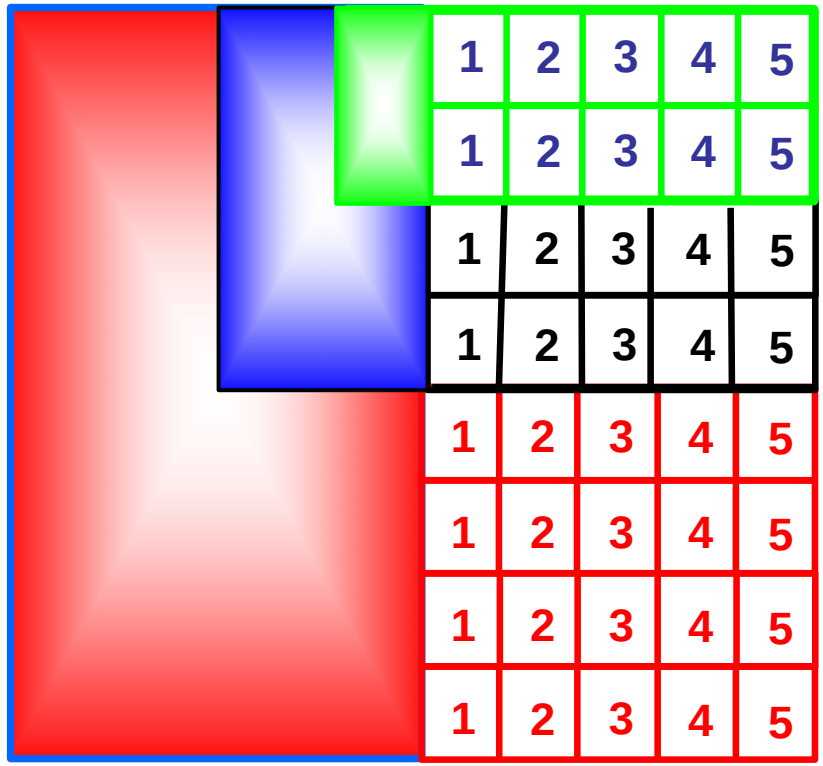
Architectural Limits and EPIC Solutions

Problem	Solution
Memory/CPU Latency is already large and growing	***Speculative Loads***
Increasing amount of conditional and/or unpredictable branches	***Predication and prediction of branches and conditionals orchestrated by the compiler to use the EPIC Architecture***
Registers and chip resource availability limit parallelism	***Increase the number of registers***
Complexity of multiple pipelines is too great for effective on chip scheduling	***Compiler handles scheduling and produces code to take advantage of the on – chip resources***

Technology case for a new Architecture

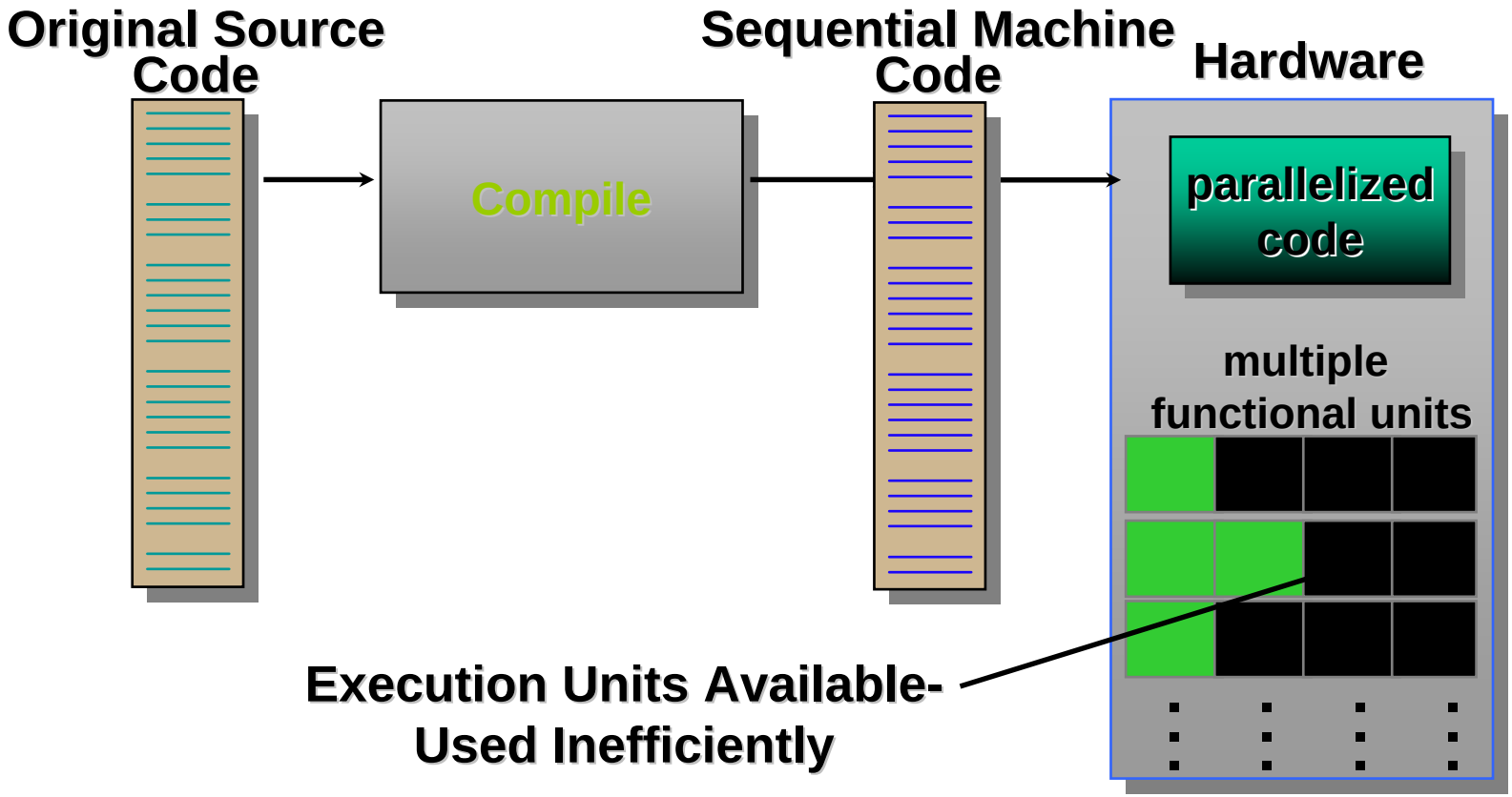
Superscalar Complexity Growth

- **Functional unit area** grows **linearly** with number of units
- **Scheduler area** grows as the **square** of the number of units



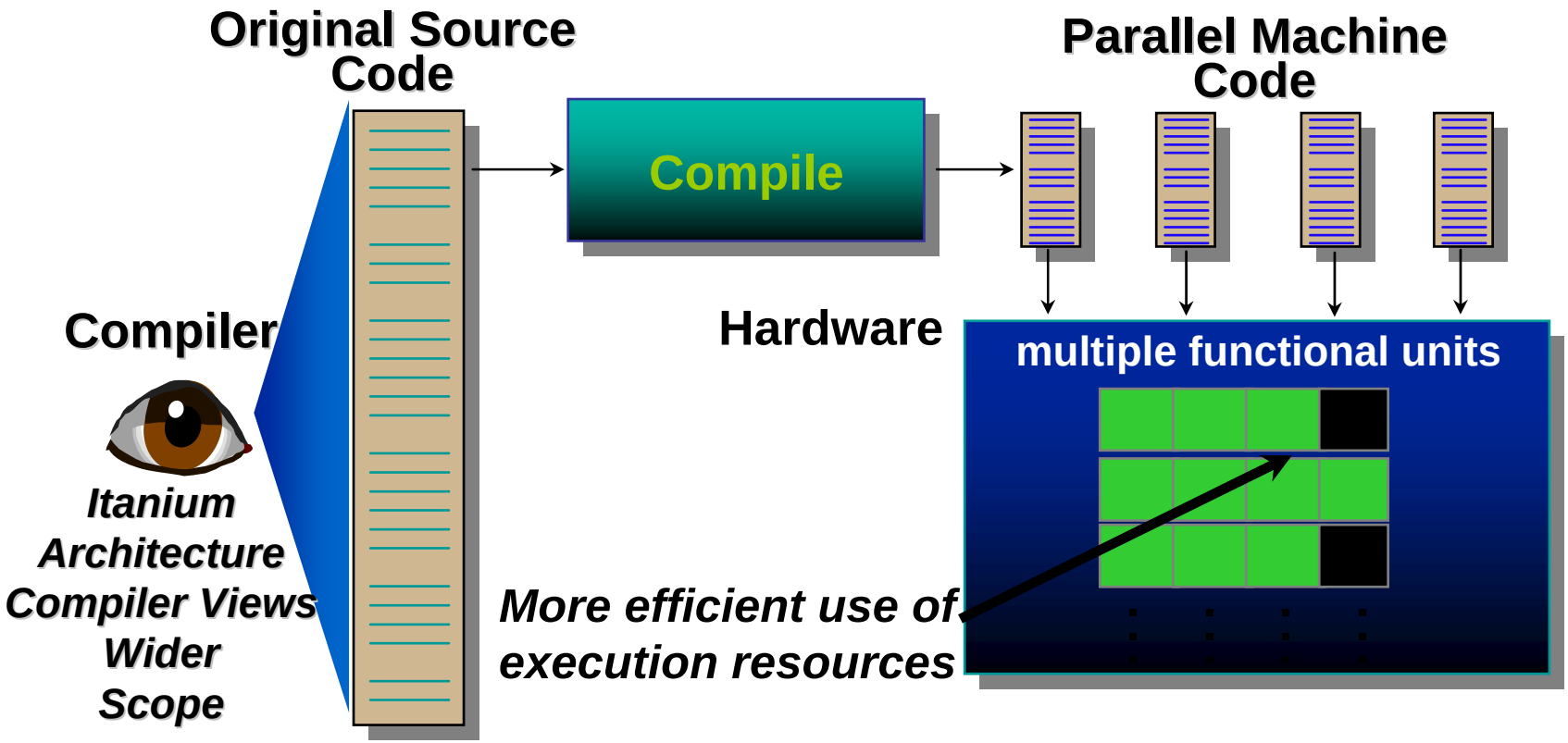
Cost-performance reaches a point of diminishing returns

Traditional Architectures: Limited Parallelism



Today's Processors are often 60% Idle

Intel® Itanium® Architecture: Explicit Parallelism



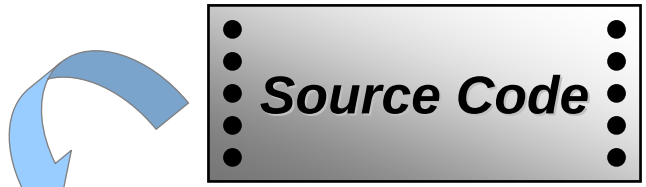
Increases Parallel Execution

Itanium® Architecture

Some Selected Features

- 64-bit Addressing Flat Memory Model
- Instruction Level Parallelism (6-way)
- Large Register Files
- Automatic Register Stack Engine
- Predication
- Software Pipelining Support
- Register Rotation
- Loop Control Hardware
- Sophisticated Branch Architecture
- Control & Data Speculation
- Powerful 64-bit Integer Architecture
- Advanced 82-bit Floating Point Architecture
- Multimedia Support (MMX™ Technology)

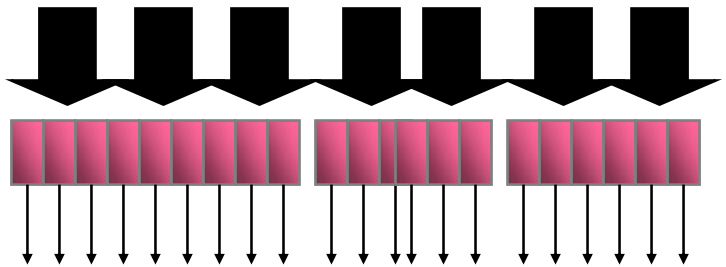
EPIC Instruction Parallelism



Instruction Groups (series of bundles)

- No RAW or WAW dependencies
- Issued in parallel depending on resources

Instruction Bundles (3 Instructions)



- 3 instructions + template
- 3 x 41 bits + 5 bits = 128 bits

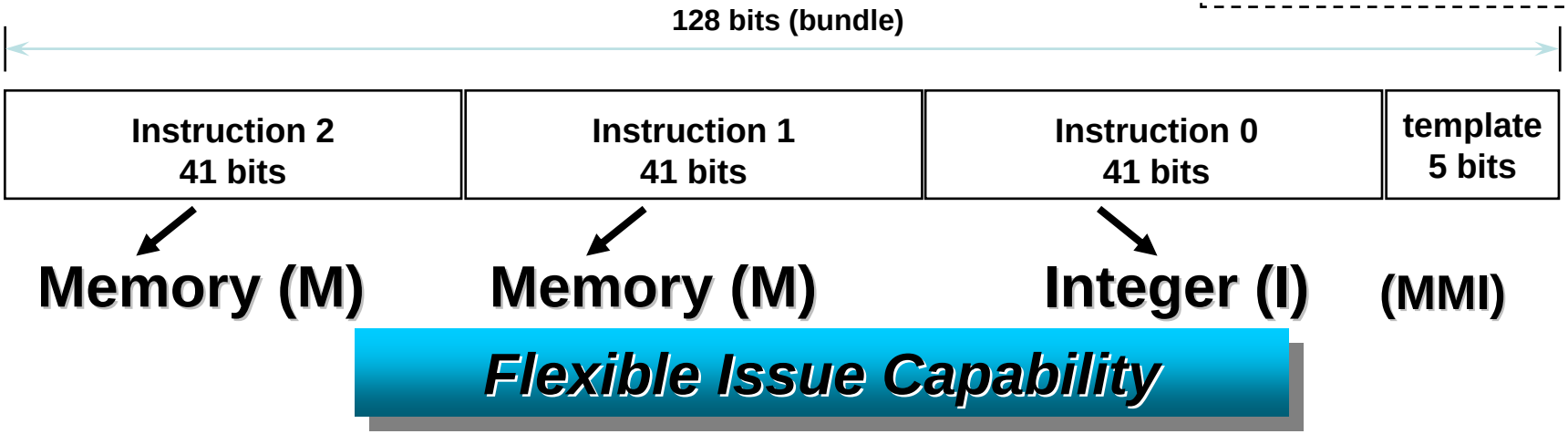
Up to 6 instructions executed per clock

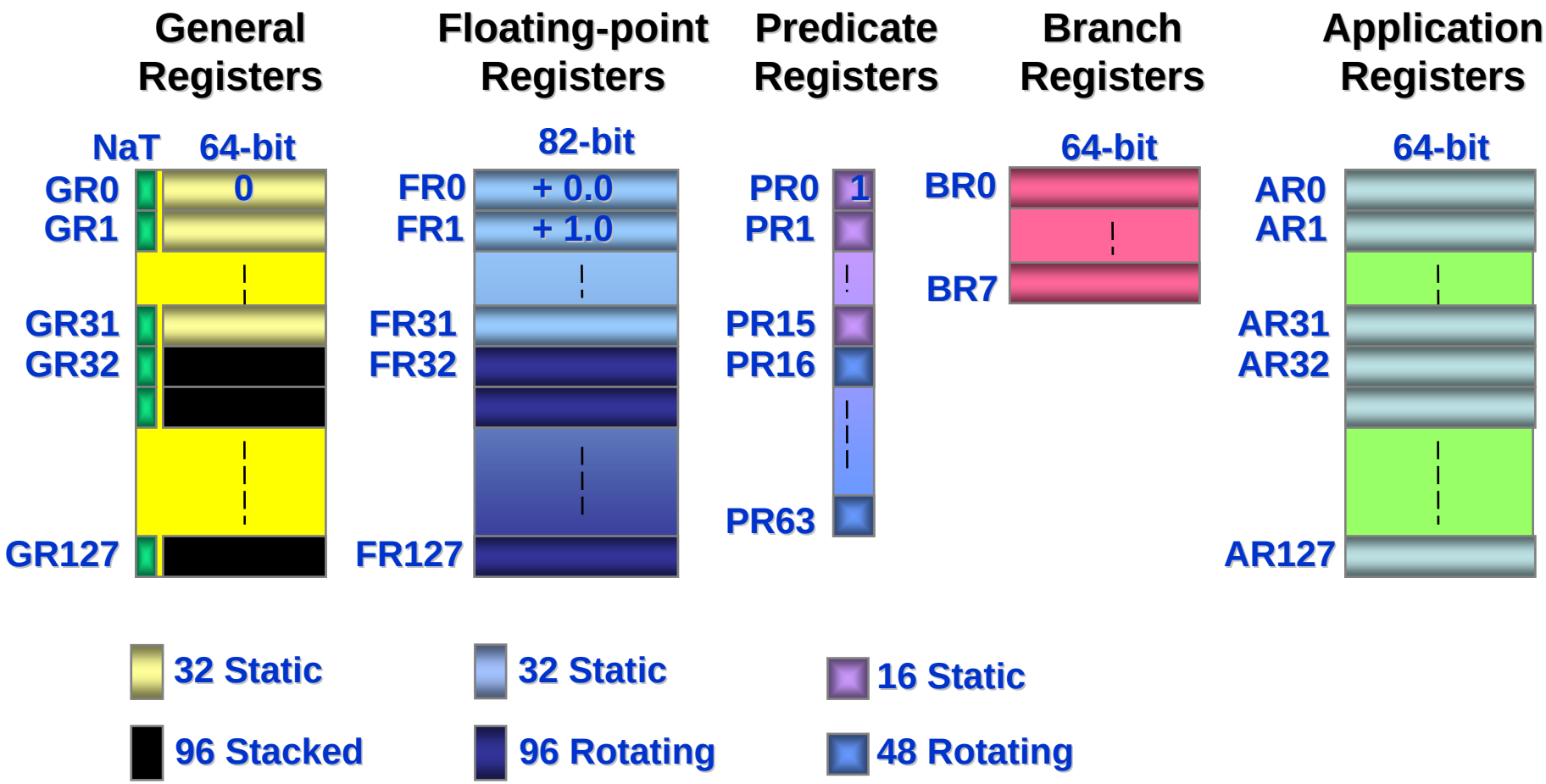
- Instruction Groups
  - No RAW or WAW dependencies
  - Delimited by ‘stops’ in assembly code
  - Instructions in groups issued in parallel, depending on available resources.

```
instr 1    // 1st. group
instr 2;;  // 1st. group
instr 3    // 2nd. group
instr 4    // 2nd. group
```

- Instruction Bundles
  - 3 instructions and 1 template in 128-bit bundle
  - Instruction dependencies by using ‘stops’
  - Instruction groups can span multiple bundles

```
{ .mii
    ld4 r28=[r8]   // load
    add r9=2,r1    // Int op
    add r30=1,r1 // Int op
}
```





Very Large Register Set

Predication

- Predicate registers activate/inactivate instructions
- Predicate Registers are set by Compare Instructions
  - Example: cmp.eq p1, p2 = r2, r3
- (Almost) all instructions can be predicated

```
(p1)    ldfd f32=[r32],8
(p2)    fmpy.d f36=f6,f36
```

- Predication:
  - eliminates branching in if/else logic blocks
  - creates larger code blocks for optimization
  - simplifies start up/shutdown of pipelined loops

Predication

- Code Example: absolute difference of two numbers

C Code

```
if (r2 >= r3)
      r4 = r2 - r3;
else
      r4 = r3 - r2;
```

Non-Predicated Pseudo Code

```
          cmpGE   r2, r3
          jump_zero P2
P1:       sub  r4 = r2, r3
          jump end
P2:       sub  r4 = r3, r2
end:      ...
```

Predicated Assembly Code

```
      cmp.ge p1,p2 = r2,r3 ;;
(p1) sub r4 = r2,r3
(p2) sub r4 = r3, r2
```

Predication Removes Branches, Enables Parallel Execution

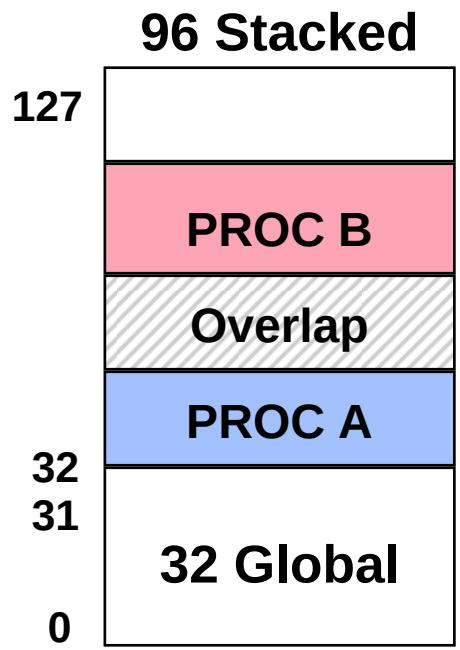
Register Stack Engine

The traditional use of a procedure stack in memory for procedure call management demands a large overhead.

The Intel® Itanium™ processor family uses the general register stack for procedure call management, thus eliminating the frequent memory accesses.

Register Stack

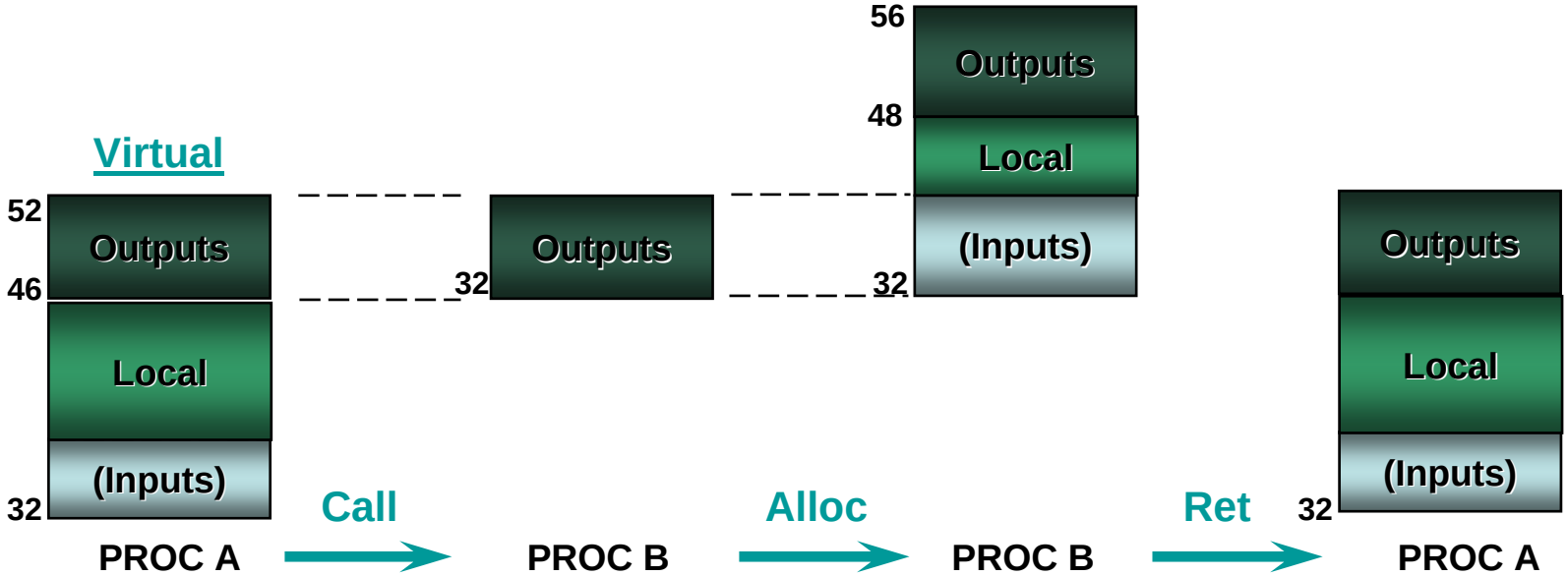
- GRs 0-31 are global to all procedures
- Stacked registers begin at GR32 and are local to each procedure
- Each procedure's register stack frame varies from 0 to 96 registers
- Only GRs implement a register stack
  - The FRs, PRs, and BRs are global to all procedures
- Register Stack Engine (RSE)
  - Upon stack overflow/underflow, registers are saved/restored to/from a backing store transparently



Optimizes the Call/Return Mechanism

Register Stack Engine at Work:

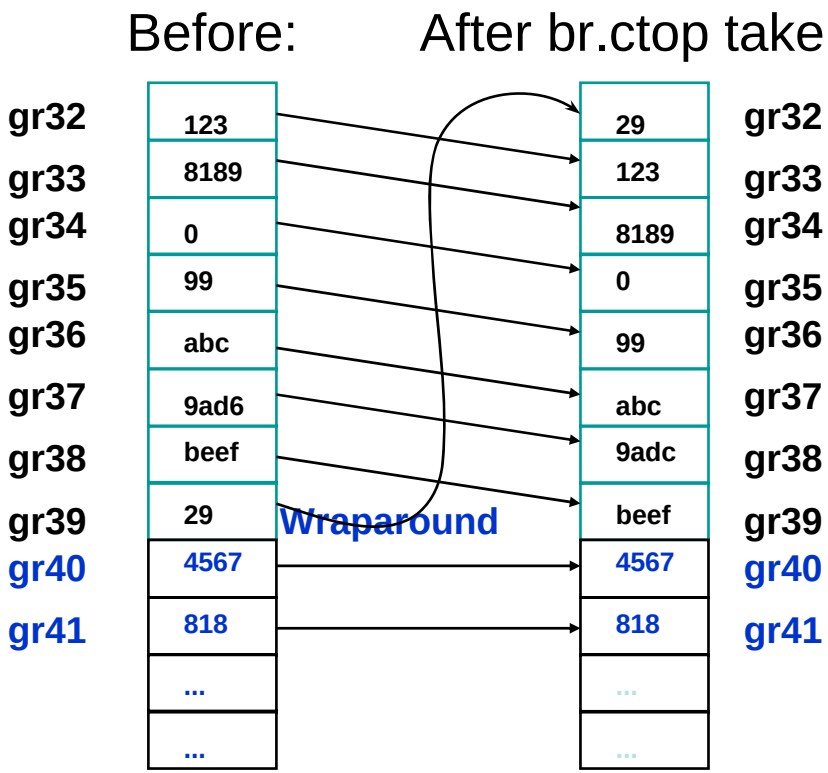
- Call changes frame to contain only the caller's output
- Alloc instr. sets the frame region to the desired size
  - Three architecture parameters: local, output, and rotating
- Return restores the stack frame of the caller



Improved execution speed in oo languages, i.e. Java, C++

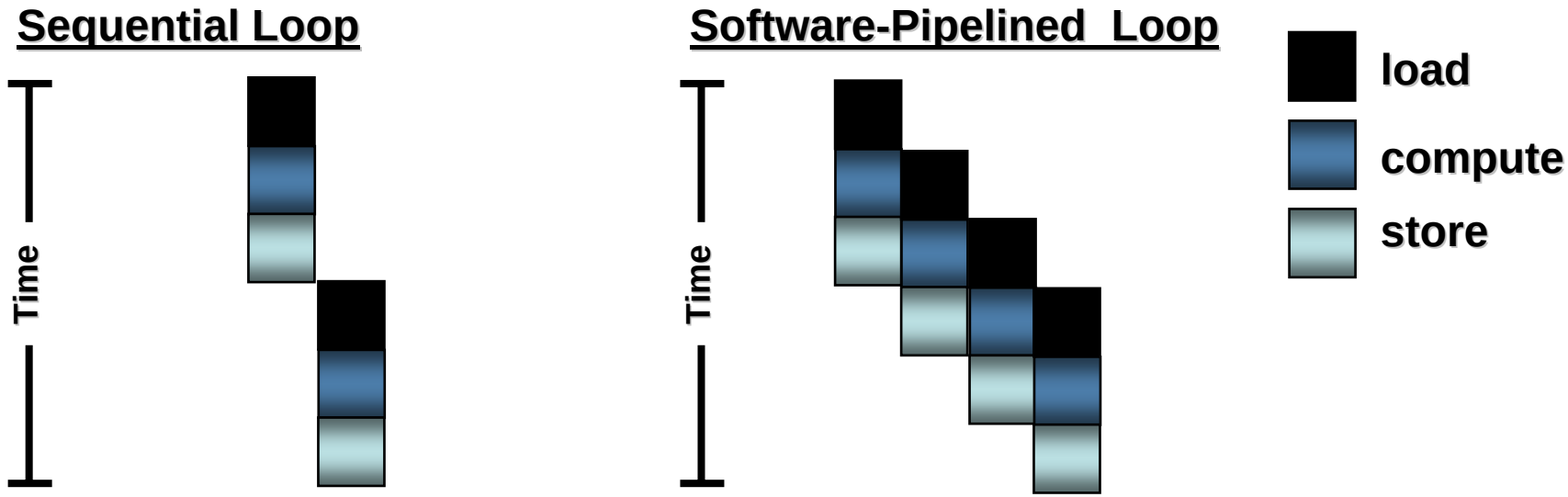
Register rotation

- Example: 8 general registers rotating, counted loop (br.ctop)



- Floating point and predicate registers
  - Always rotate the same set of registers
    - FR 32-127
  - Rotate in the same direction as general registers
    - highest rotates to lowest register number
    - all other values rotate towards larger register numbers
  - Rotate at the same time as general registers (at the modulo-scheduled loop instruction)

Software Pipelining



- Traditional architectures use loop unrolling
  - Results in code expansion and increased cache misses
- Itanium™ Software Pipelining uses rotating registers
  - Allows overlapping execution of multiple loop instances

Itanium™ provides direct support for Software Pipelining

Software pipelined Loop

- Consider

C code:
```
for (i = 0; i < n; i++)
        y[i] = a * x[i];
```

Pseudo Code:
```
loop:   ldfd x[i]
        fmpy.d y[i] = a, x[i]
        stfd y[i]
        br.ctop loop
```

- Assume
  - Instruction Latencies:
    - ldfd (fp load) 4 cycles*
    - fmpy.d (fp mul) 2 cycles*
    - stfd (fp store) 1 cycle*
    - br.ctop (branch counted loop top) 1 cycle*
  - ldfd, fmpy.d, stfd and br can be issued in the same instruction group (only w/o RAW or WAW dependencies)

*Cycle counts for demonstration purposes only.

Software pipelined loop

Cycle 1:	ld x[1]			
Cycle 2:	ld x[2]			
Cycle 3:	ld x[3]			
Cycle 4:	ld x[4]			
Cycle 5:	ld x[5]	fmpy y[1]=a,x[1]		
Cycle 6:	ld x[6]	fmpy y[2]=a,x[2]		
Cycle 7:	ld x[7]	fmpy y[3]=a,x[3]	stf y[1]	
Cycle 8:	ld x[8]	fmpy y[4]=a,x[4]	stf y[2]	
Cycle 9:		fmpy y[5]=a,x[5]	stf y[3]	
Cycle 10:		fmpy y[6]=a,x[6]	stf y[4]	
Cycle 11:		fmpy y[7]=a,x[7]	stf y[5]	
Cycle 12:		fmpy y[8]=a,x[8]	stf y[6]	
Cycle 13:			stf y[7]	
Cycle 14:			stf y[8]	

For n = 8

Prolog

Kernel

Epilog

** Cycle counts for demonstration purposes only.*

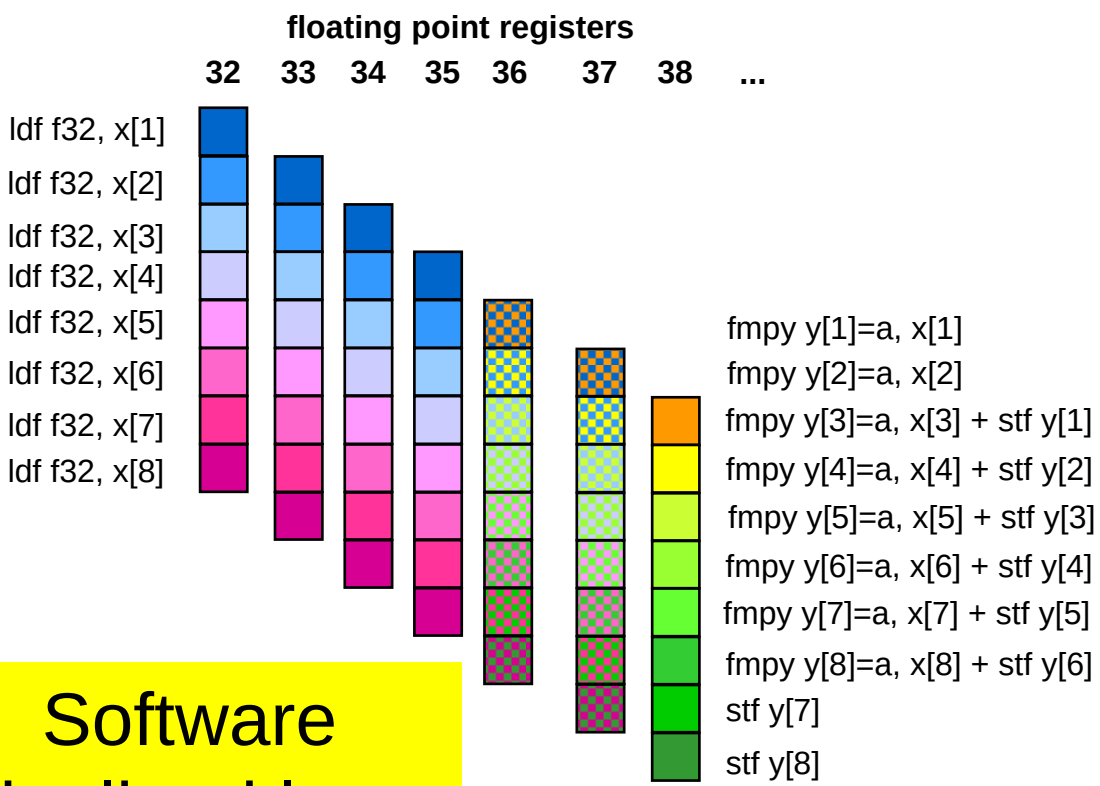
In this example, one iteration takes 7 cycles

Loop body:

```
(p16) ldf f32,[r32],8
(p20) fmpy f36=f6,f36
(p22) stf [r33],f38,8
      br.ctop
```

Remember our example latencies:

ldf	4 cycles
fmpy	2 cycles
stf	1 cycle
br	1 cycle



predicate registers

16	17	18	19	20	21	22	...
1	0	0	0	0	0	0	
1	1	0	0	0	0	0	
1	1	1	0	0	0	0	
1	1	1	1	0	0	0	
1	1	1	1	1	0	0	
1	1	1	1	1	1	0	
1	1	1	1	1	1	1	
1	1	1	1	1	1	1	
0	1	1	1	1	1	1	
0	0	1	1	1	1	1	
0	0	0	1	1	1	1	
0	0	0	0	1	1	1	
0	0	0	0	0	1	1	
0	0	0	0	0	0	1	

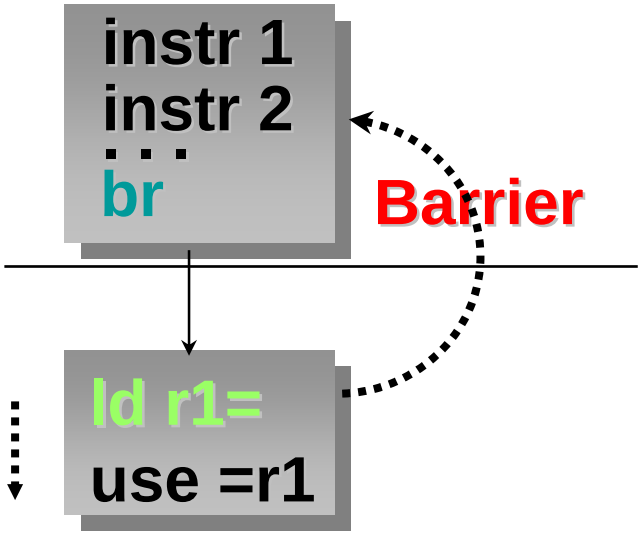
Software pipelined loop

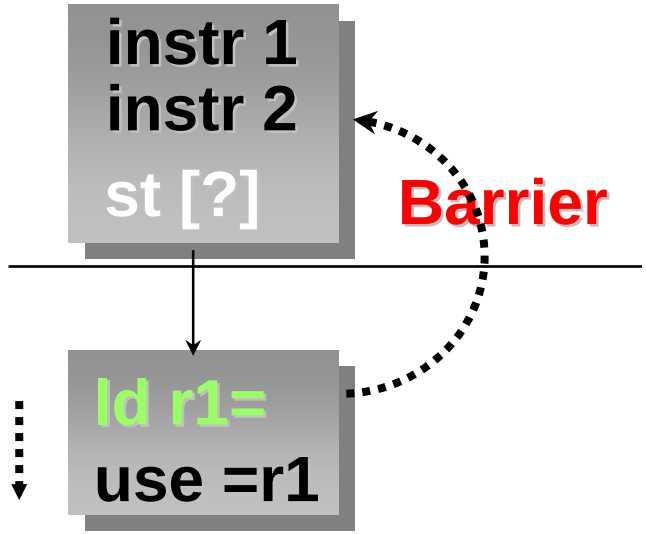
Software pipelined loop

- *Actual code example:*

```
                // Initialization
                mov pr.rot=0                    // Clear all rotating predicates
                cmp.eq p16,p0=r0,r0             // Set p16=1
                mov ar.lc=7                     // Set loop counter to n-1
                mov ar.ec=7                     // Set epilog counter # of stages
                ...
loop:
                { .mfi
(p16)           ldfd f32=[r32],8                // Stage 1: Load x
(p20)           fmpy.d f36=f6,f36               // Stage 5: y=a*x
                nop.i 0
                }
                { .mfb
(p22)           stfd [r33]=f38,8                // Stage 7: Store y
                nop.f 0
                br.ctop.sptk.few  loop          // Branch back
                }
```

Control & Data Speculation

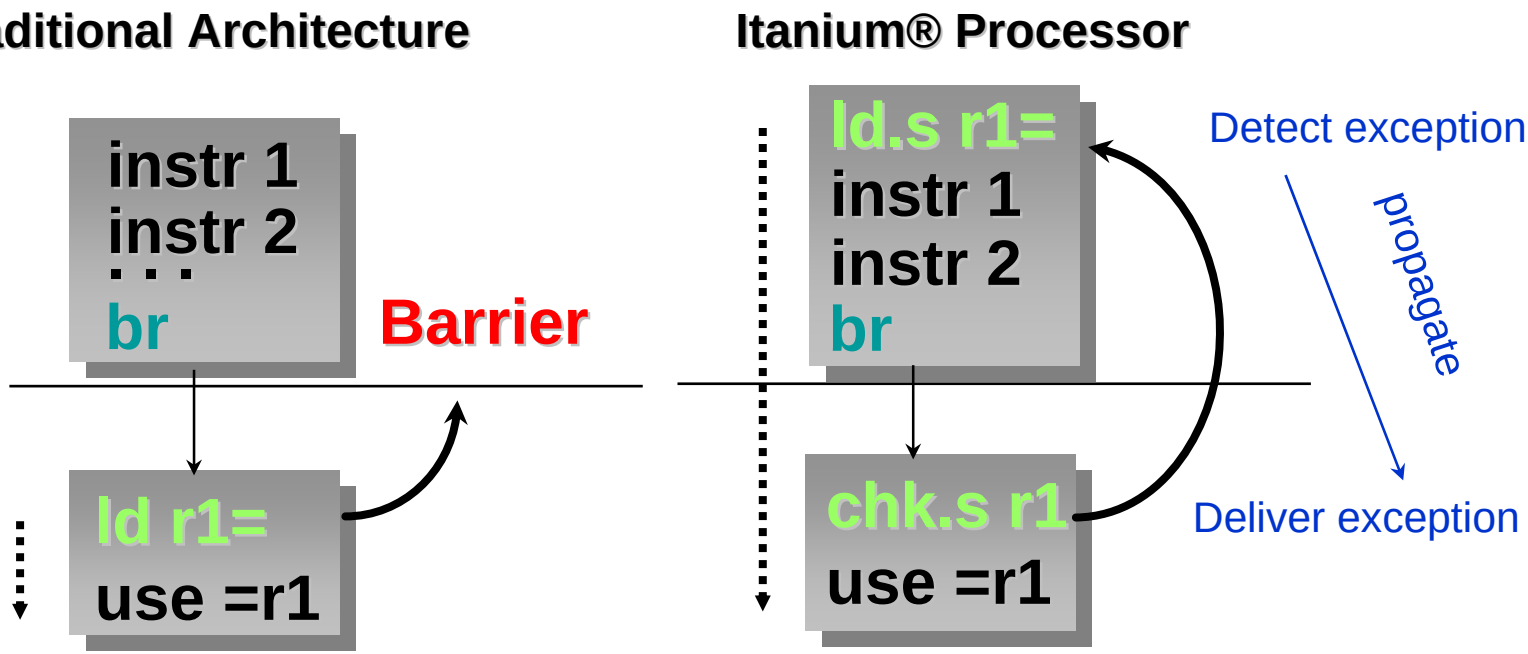




- Control Speculation moves loads above branches / calls

- **Data Speculation moves loads above possibly conflicting stores**

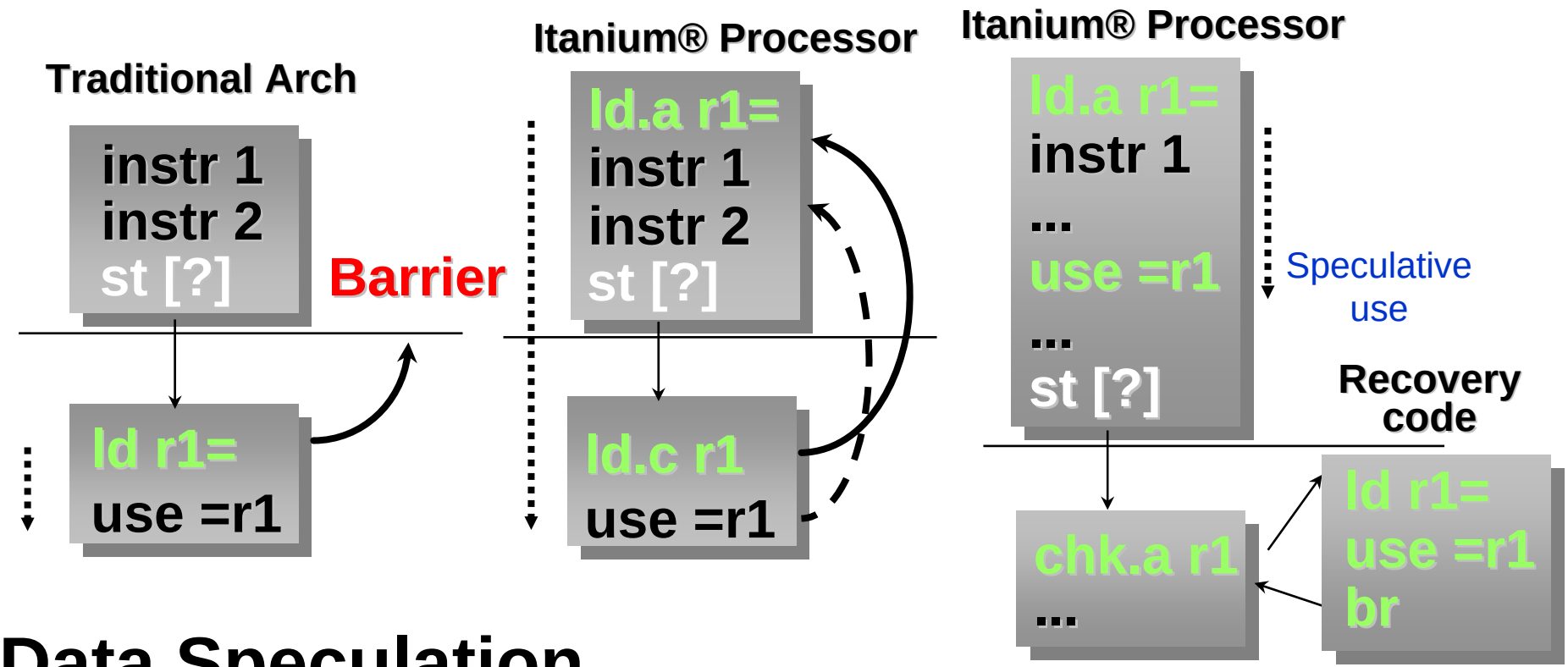
Speculation reduces the impact of memory latency



Control Speculation:

- Control Speculation moves loads above branches
  - Detected exception indicated using NaT bit / NaTVal
- Check raises detected exceptions

Barrier bridged, memory latency masked

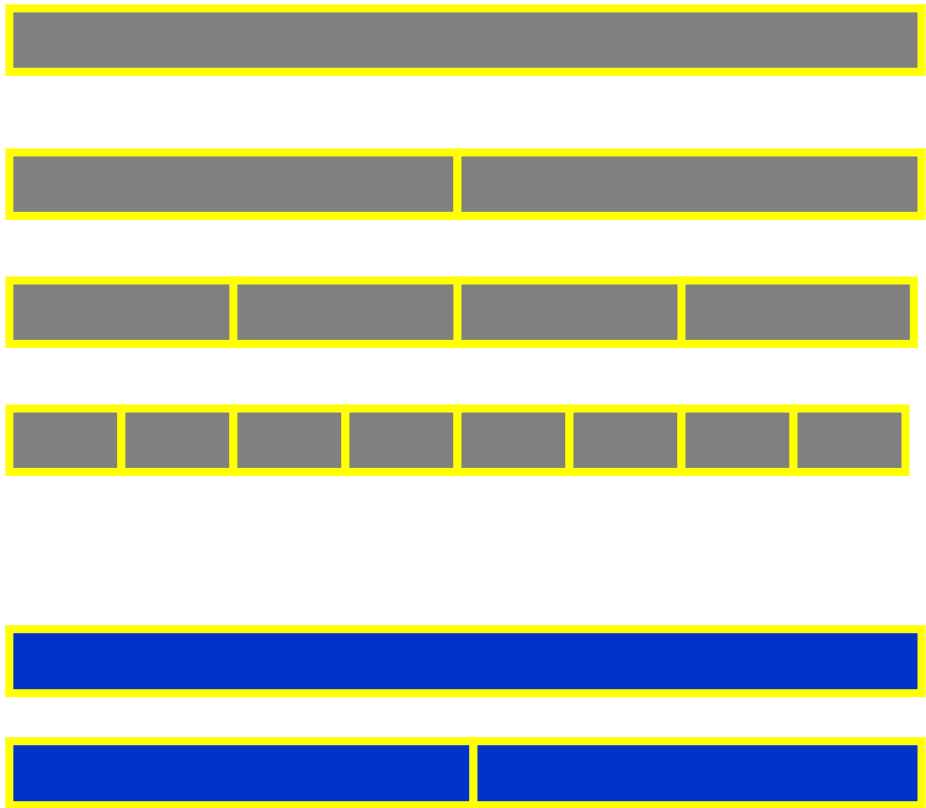


Data Speculation

- Data Speculation moves loads above possibly conflicting stores
  - Keeps track of load addresses used in advance (ALAT)
- Advanced-loaded data can be used speculatively

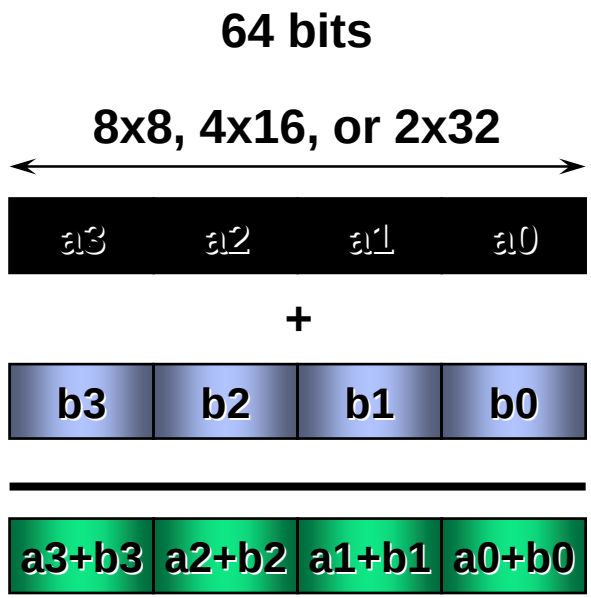
Data and Control Speculations can be combined

Itanium® HW Data Types



SIMD - Integer

- Exploits data parallelism with SIMD (*Single Instruction Multiple Data*)
- Performance boost for audio, video, imaging etc. functions
- GRs treated as 8x8, 4x16, or 2x32 bit elements
- Several instruction types
  - Addition and subtraction, multiply
  - Pack/Unpack
  - Left shift, signed/unsigned right shift
- Compatible with Intel MMX™ Technology

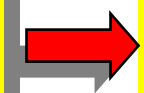


Available through Compiler Intrinics

Floating-Point Architecture

- 128 Floating Point registers (82 bit)
  - Single, double, double-extended data types
- Full IEEE.754 compliance
- Arithmetic
  - FMA – Multiply-Add instruction f = a * b + c
  - SW Divide / Sqrt, provide high throughput, take advantage of wide FP machine
  - Max, Min instructions for Floating-point
- Data transfer
  - load, store, GR ⇔ FR conversion; load pair to double data

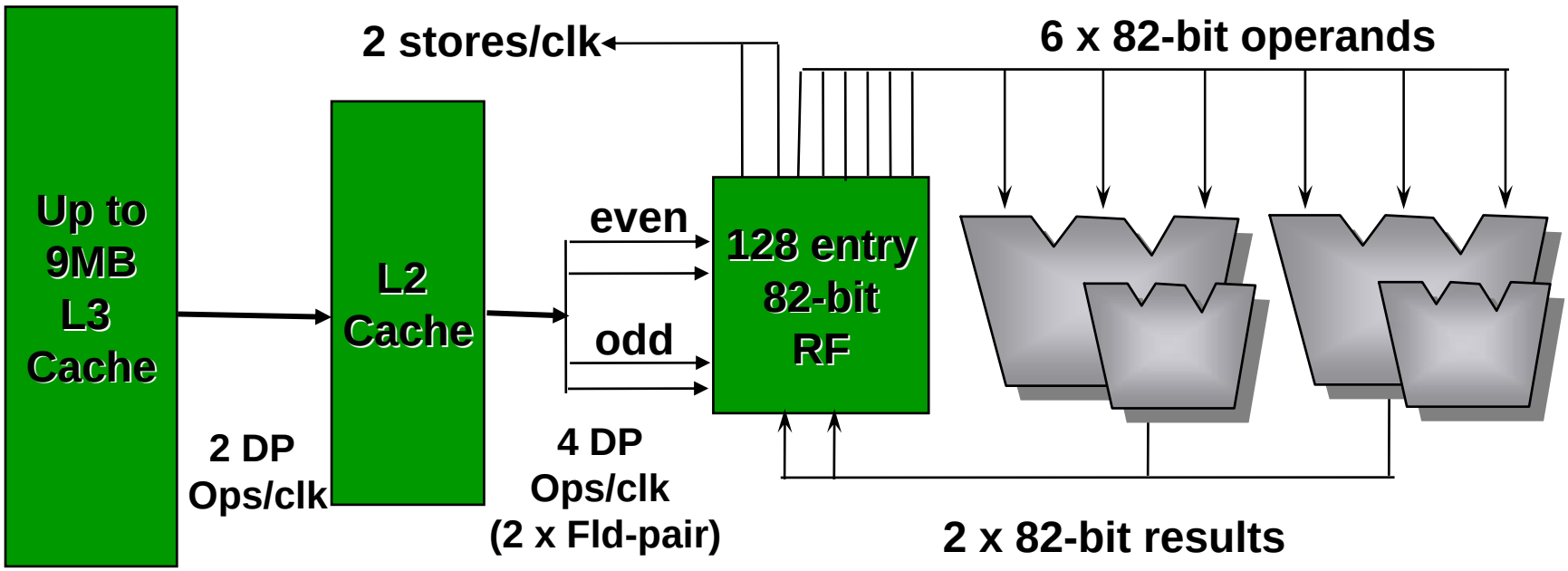
2 independent FP Units
Up to 4 DP FP operations per clock
Up to 4 DP FP operands loaded per clock



Excellent Workstation/3D Apps Performance

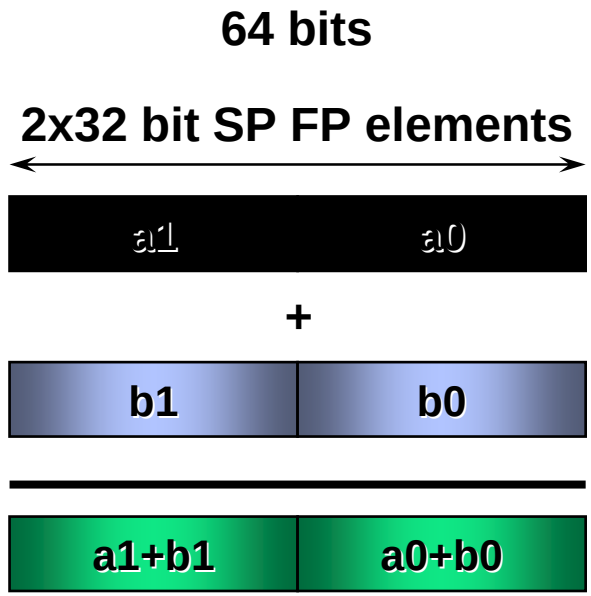
Floating Point Features

- Native 82-bit hardware provides support for multiple numeric models
- 2 Extended precision pipelined FMACs deliver 4 EP / DP FLOPs/cycle
- Performance for security, efficient use of hardware: Integer mul-add, s/w divide
- Balanced with plenty of operand bandwidth from registers / memory



SIMD - Floating-Point

- Exploits data parallelism with SIMD (*Single Instruction Multiple Data*)
- Up to 2x performance boost
- FP Registers treated as two 32 bit single precision elements
  - –Full IEEE.754 compliance
  - –Availability of fast divide (non IEEE)
- Compatible with Streaming SIMD Extensions (SSE)



Enables World Class 3D Graphics Performance

Some Floating Point Latencies

Operation	Latency
FP Load (L2 Cache hit)	6
FMAC,FMISC	4
FP -> Int (getf)	5
Int -> FP (setf)	6
Fcmp to branch Fcmp to qual pred	2 2

L1D Cache (Integer-only)

- High Performance 16GB/s, 2 ld AND 2 st ports
  - **16KB, 64 byte cache line, 4-way**
  - **Not used for FP data**
  - **Write Through – all stores are pushed to the L2**
  - **True dual-ported read access – no load conflicts**
  - **pseudo-dual store port write access**
    - 2 store coalescing buffers/port hold data until L1D update
  - **Store to load forwarding**

One clock data cache provides a significant performance benefit

L2 and L3 Cache

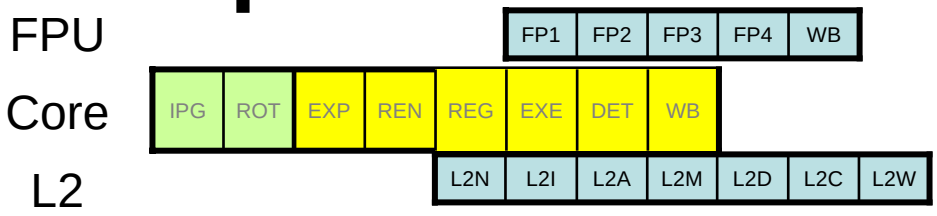
- L2 256KB, 32GBs, 5-7 clk
  - Data array is banked - 16 banks of 16KB each
- Non-blocking / out-of-order
  - L2 queue (32 entries) - holds all in-flight load/stores
  - out-of-order service - smoothes over load/store/bank conflicts, fills
  - Can issue/retire 4 stores/loads per clock
  - Can bypass L2 queue (5,7,9 clk bypass) if
    - » no address or bank conflicts in same issue group
    - » no prior ops in L2 queue want access to L2 data arrays
- Up tp 9MB L3, 32GBs,12-13 clk cache on die!!
  - Single ported – full cache line transfers

Itanium® 2 Caches

	L1I	L1D	L2	L3
Size	16K	16K	256K	1.5/3/6M/9M on die
Line Size	64B	64B	128B	128B
Ways	4	4	8	12
Replacement	LRU	NRU	NRU	NRU
Latency (load to use)	I-Fetch:1	INT:1	INT: 5 FP: 6	12/13
Write Policy	-	WT (RA)	WB (WA+ RA)	WB (WA)
Bandwidth	R: 32 GBs	R: 16 GBs W: 16 GBs	R: 32 GBs W: 32 GBs	R: 32 GBs W: 32 GBs

All caches are pipelined, and non-blocking

Itanium® 2 Pipelines



IPG	IP Generate, L1I Cache and TLB access		EXE	ALU Execute(6), L1D Cache and TLB access + L2 Cache Tag Access
ROT	Instruction Rotate and Buffer		DET	Exception Detect, Branch Correction
EXP	Expand, Port Assignment and Routing		WB	Writeback, Integer Register update
REN	Integer and FP Register Rename		FP1-WB	FP FMAC pipeline + reg write
REG	Integer and FP Register File read		L2N-L2I	L2 Queue Nominate/Issue
			L2A-W	L2 Access, Rotate, Correct, Write

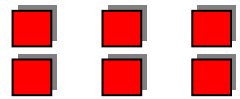
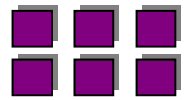
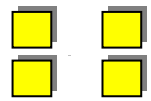
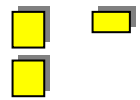
Short 8-stage in-order main pipeline

- In-order issue, out-of-order completion
- Reduced branch misprediction penalties

Pipelines are designed for very low latency

Functional Units

- Integer
  - 6 ALUs
  - 6 Multi-Media
- Memory Ports
  - 2 Load, 4 FP Load
  - 2 Store
- Floating Point
  - 2 MACs
- Branch Ports
  - 3 Branches

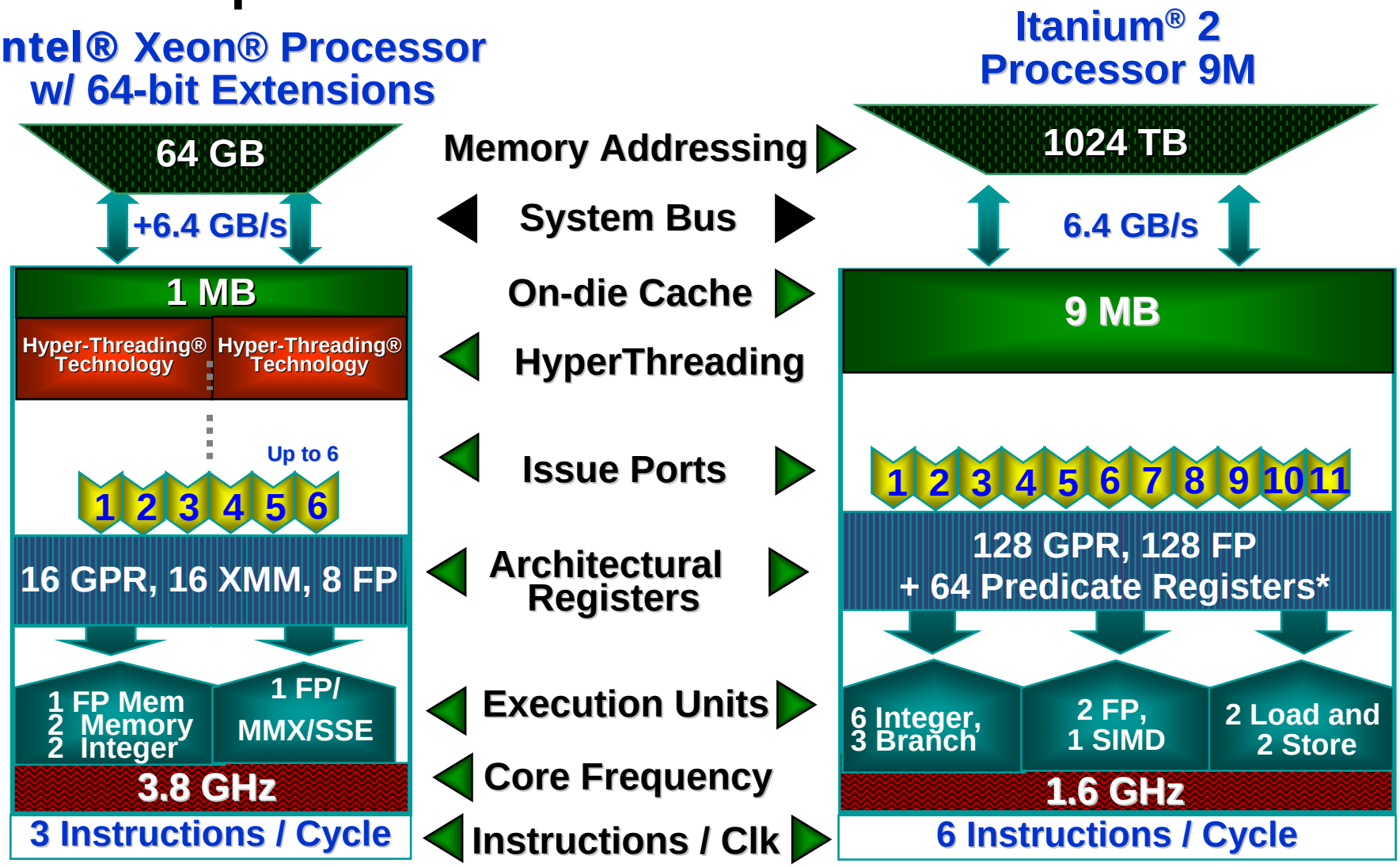
	Itanium 2 Processor
Integer	
FP - 64/82bit	
FP - 32bit (SIMDFP)	
Multimedia	
Load/Store	
Branch	

TLBs

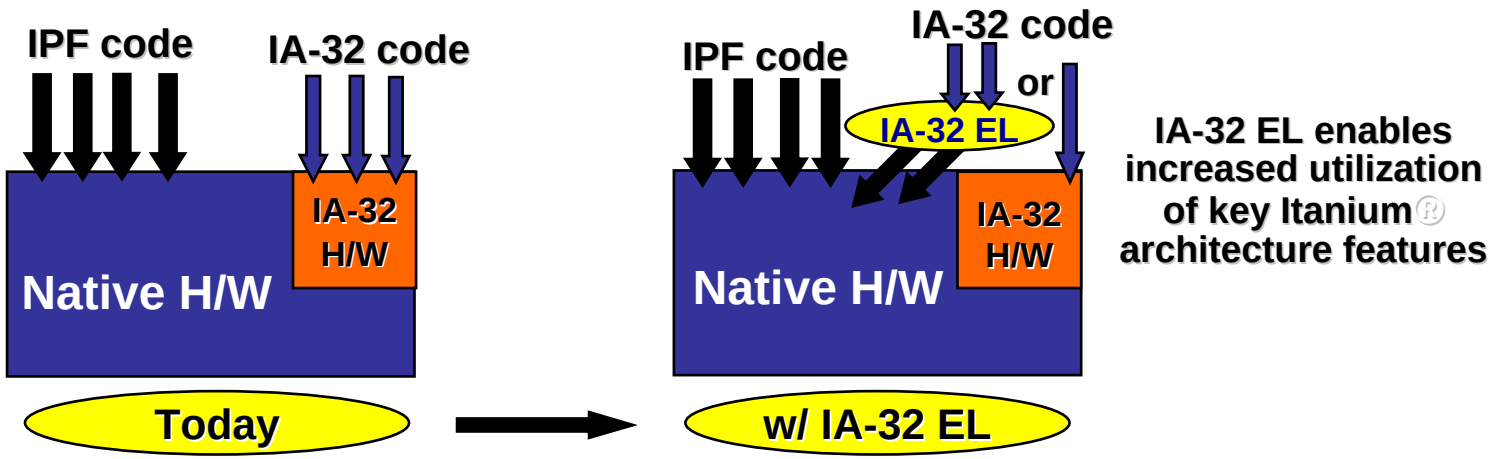
- 2-level TLB hierarchy
  - DTC/ITC (32/32 entry, fully associative, 0.5 clk)
    - Small fast translation caches tied to L1D/L1I
      - Key to achieving very fast 1-clk L1D, L1I cache accesses
  - DTLB/ITLB (128/128 entry, fully associative, 1 clk)
    - All architected page sizes (4K to 4GB)
    - Supports up to 64/64 ITR/DTRs
    - TLB miss starts hardware page walker

Small fast TLBs enable low latency caches

Intel Enterprise Micro-Architectures

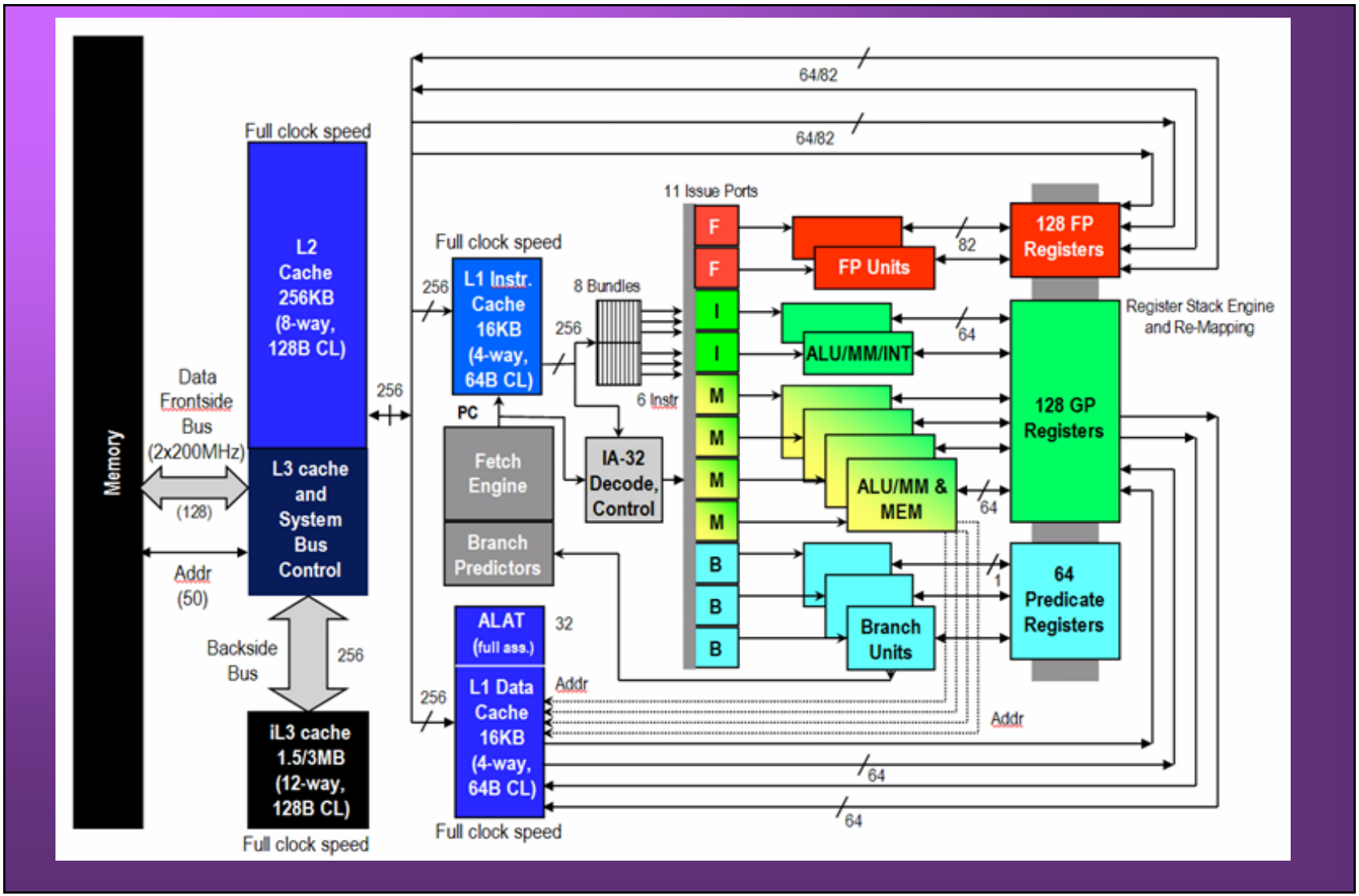


IA-32 App Support on Itanium®



- IA-32 applications are supported on all Itanium® processor offerings
  - Before: IA-32 hardware-based approach
  - Since 2004: IA-32 Execution Layer enhances 32-bit application support
- IA-32 Execution Layer – What is it?
  - Translates the IA-32 application code into native Itanium code “just-in-time”
- Benefits of IA-32 EL
  - Enables the use of new IA-32 instructions (e.g. SSE2)
  - Can accelerate IA-32 applications on Itanium®-based systems
- Available for all major OS distributions today
  - Microsoft Windows, SUSE & Redhat Linux

Itanium® 2 Block Diagram



Some notes on Intel® Itanium® Architecture Application Programming Model

Introduction to Basic Itanium® Processor Instructions

- High-Level introduction to basic instruction set
- Examples, not comprehensive
- Important to be able to read, understand and to evaluate quality of compiler generated code
- Helpful to understand debugger and performance analyzer displays
- No requirement to do assembly programming

In fact, hand coding for Itanium processors is discouraged!

Basic Instruction Format

[(qp)] mnemonic[.comp] dest = srcs

where:

***qp** = qualifying predicate.*

***mnemonic** = unique name identifying the instruction.*

***comp** = one or more completers.*

***dest** = destination operand(s).*

***srcs** = source operand(s).*

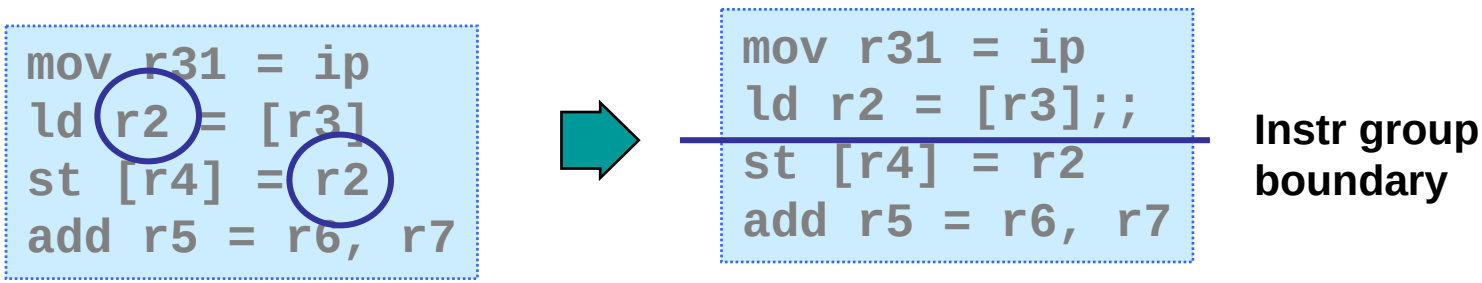
Example: **(p10) ld4.s r31 = [r3]**

Instruction Groups

- Instruction group dependency violations

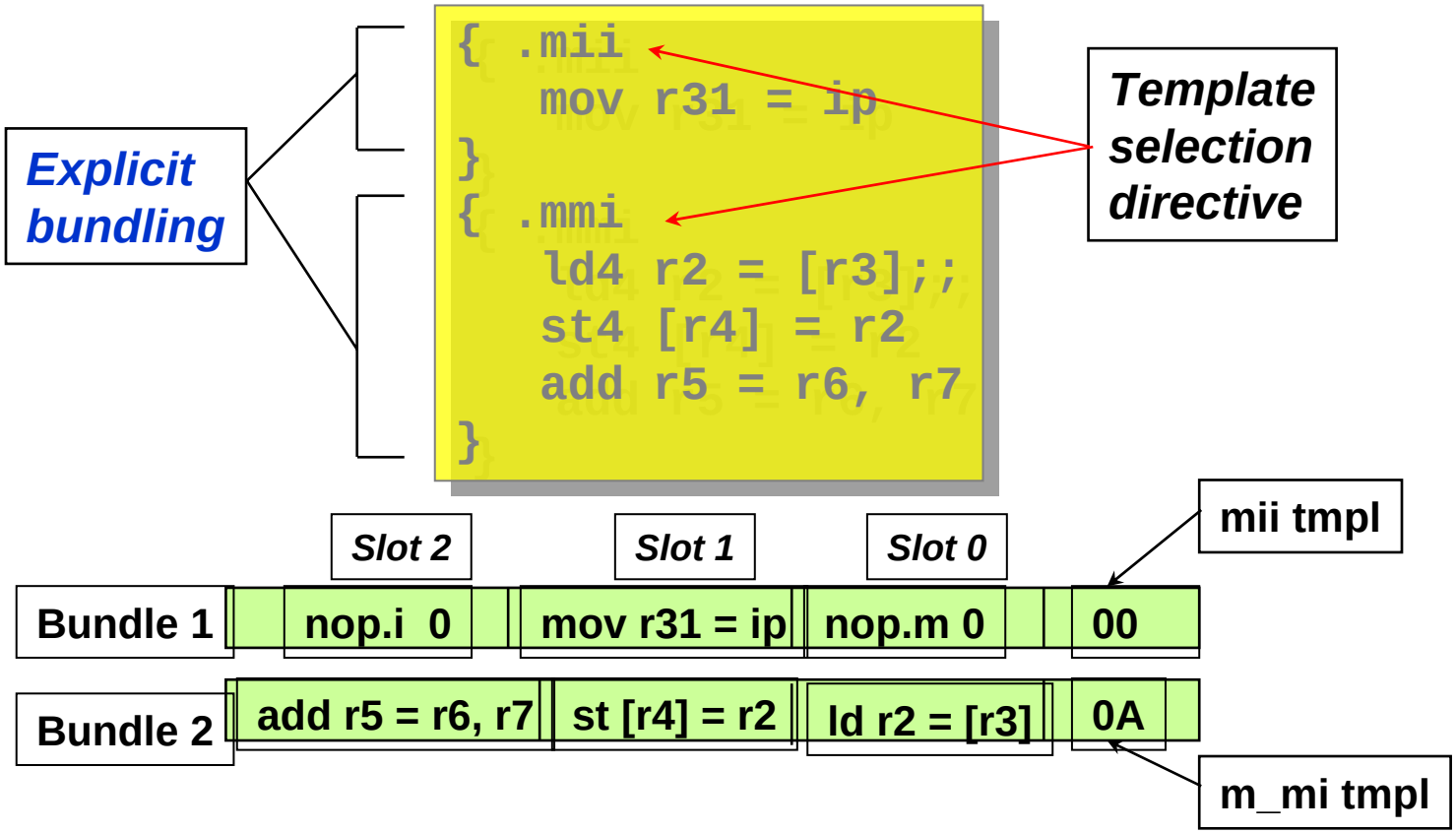
	PERMITTED within an instr group	NOT PERMITTED§ within an instr group
Register	Write-After-Read (WAR)	Read-After-Write (RAW)§§ Write-After-Write (WAW)§§
Memory/ ALAT	RAW, WAW, WAR	

§*Not permitted is used in the context that it will result in undefined behavior.*
§§*See notes for RAW and WAW exceptions to these rules.*



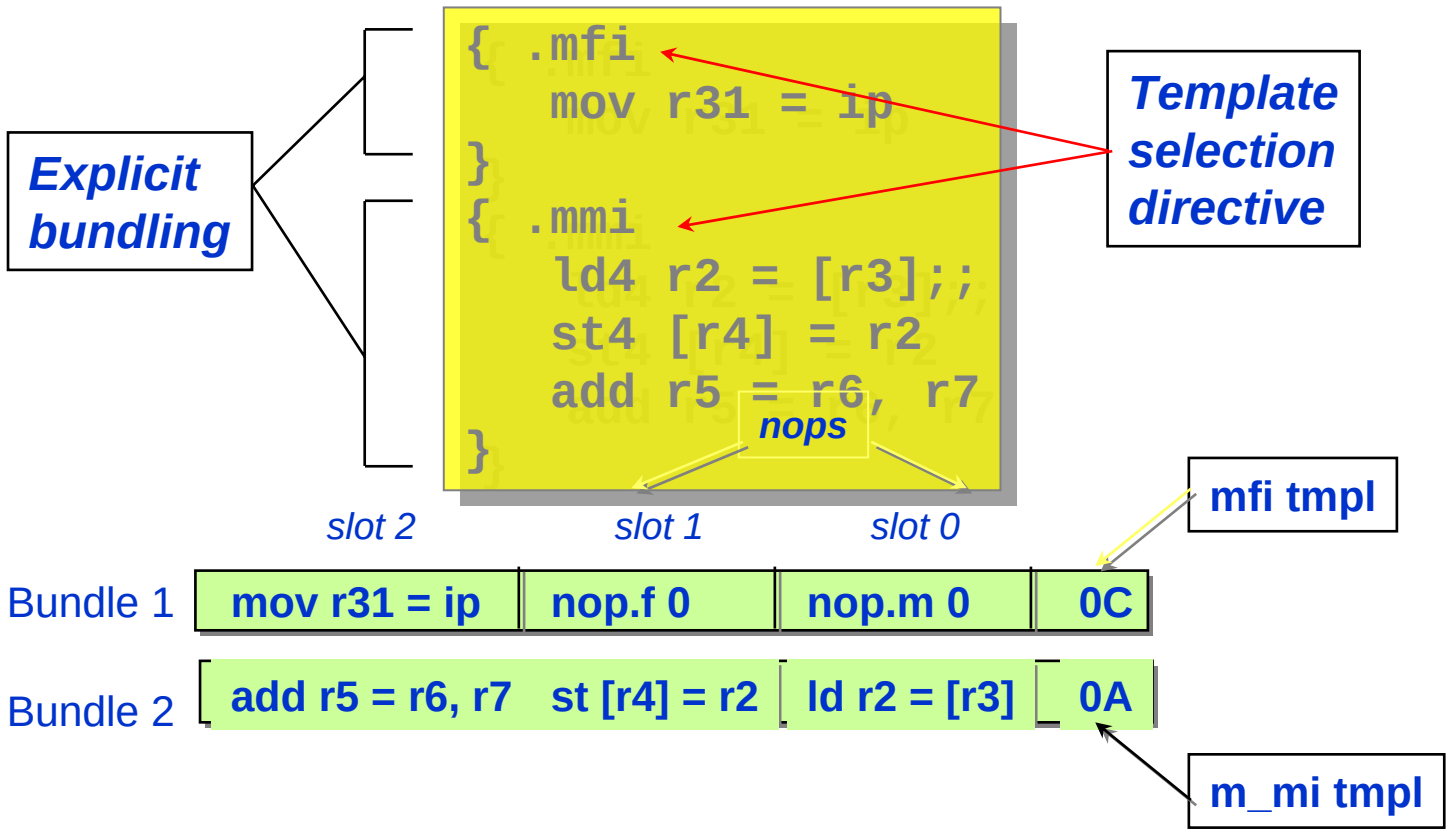
Assembly Language Features

- Bundling: One example



Assembly Language Features

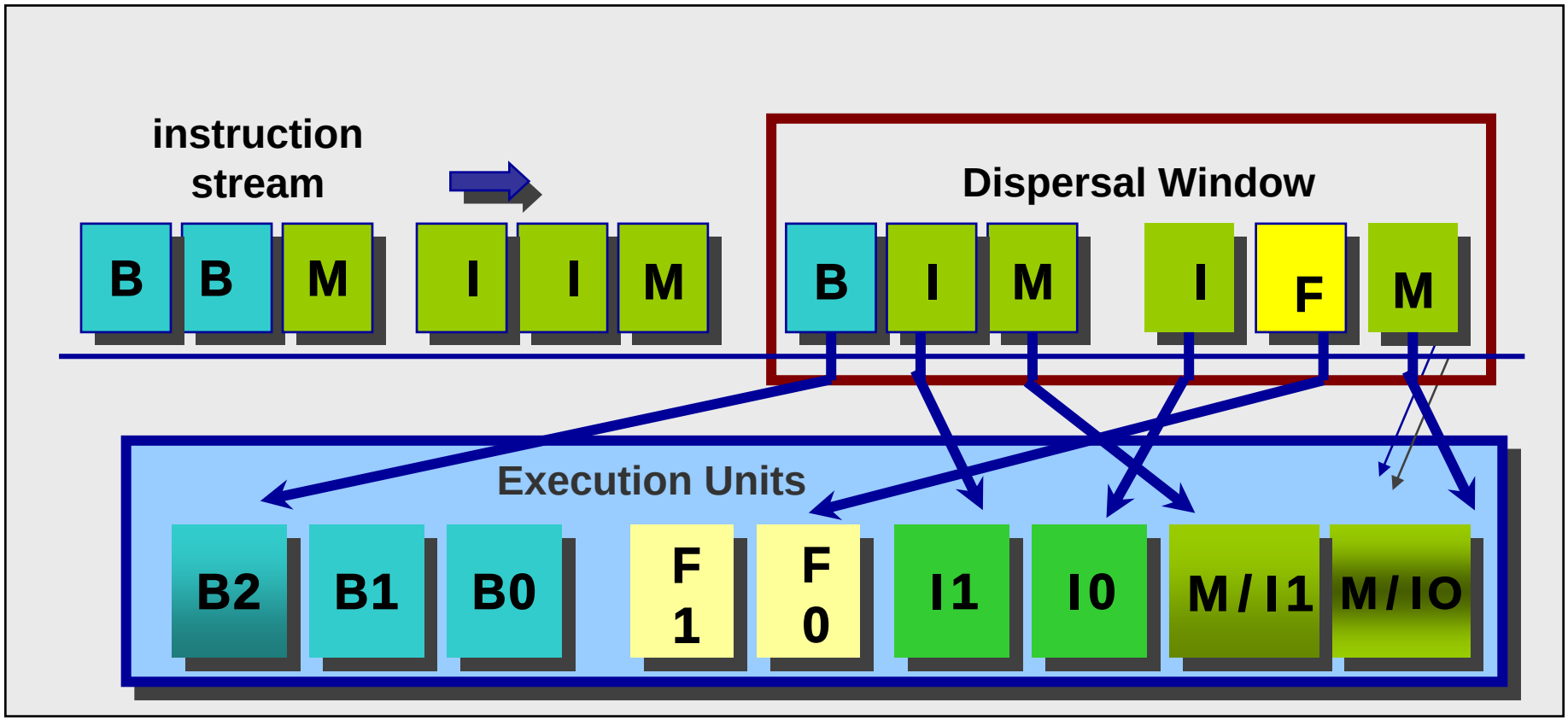
- Same code sequence but different bundling



Assembly Language Features

- The very same action can be implemented by multiple, different bundle combinations in general
- This is key to Itanium® architecture performance.
- It is the job of the compiler/ programmer to find the optimal combinations of bundles for maximum performance!

Instruction Dispersal, Itanium® 2 Implementation



Flexible Issue Capability

Up to 6 instructions executed per clock

Dispersal Matrix – Itanium® 2 Processor

	MII	MLI	MMI	MFI	MMF	MIB	MBB	BBB	MBB	MFB
MII	*		*	*	*	*	X	X	*	X
MLI	*	*	*	X	*	X	*	X	*	X
MMI	*	*	*	*	*	*	*	X	*	*
MFI	*	X	*	X	*	X	X	X	*	X
MMF	*	*	*	*	*	*	*	X	*	*
MIB§	*	X	*	X	*	X	X		*	X
MBB										
BBB										
MMB§	*	*	*	*	*	*	*		*	*
MFB§	X	X	*	X	*	X	X		*	X

§Hint in first bundle

* **Possible Itanium 2 processor full issue**

X **Possible Itanium 2 and Itanium® processor full issue**

Architectural Improvements Allow Faster Execution Through More Issued Instructions/Cycle

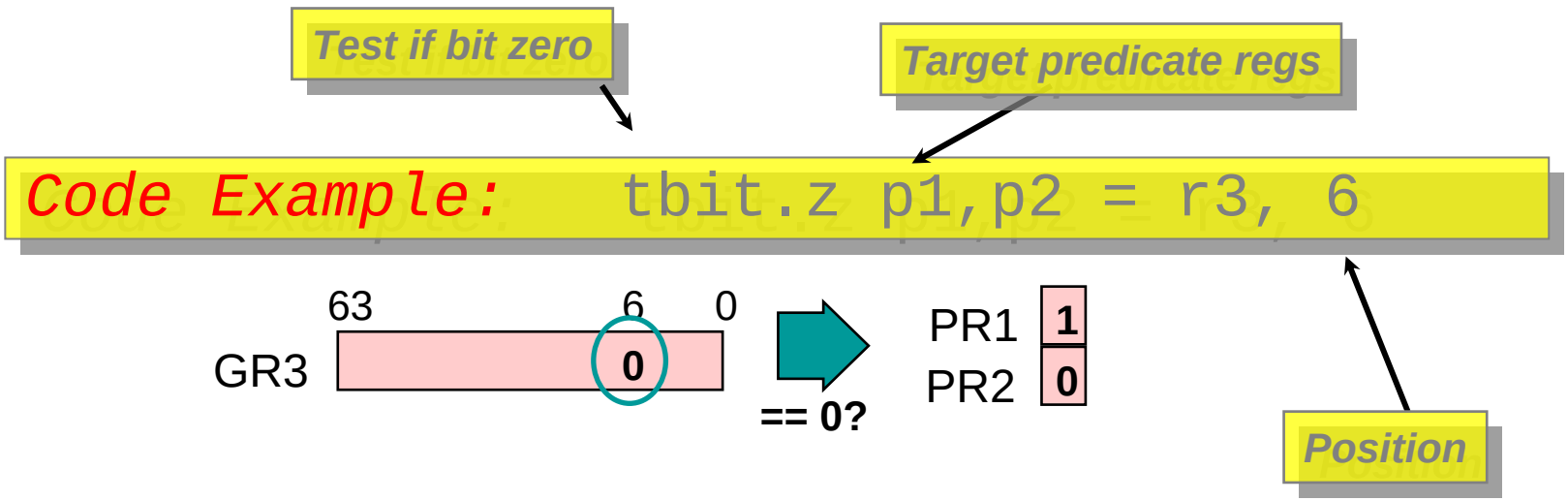
Register Symbol Names

- ***General Registers***
  - fixed regs **r0-r31**
  - stacked regs **r32-r127**
  - *input regs* ***in0-in95***
  - *local regs* ***loc0-loc95***
  - *output regs* ***out0-out95***
  - *global pointer (r1)* ***gp***
  - *return value regs* ***ret0-ret3***
  - *stack pointer (r12)* ***sp***
- ***Floating-Point Registers***
  - floating-point regs **f0-f127**
  - *argument register* ***farg0-farg7***
  - *return value regs* ***fret0-fret7***
- ***Predicate Registers***
  - predicate regs **p0-p63**
  - *all predicates* ***pr***
  - *rotating regs* ***pr.rot***
- ***Branch Registers***
  - branch regs **b0-b7**
  - *return pointer (b0)* ***rp***
- ***Application Registers***
  - app regs **ar0-ar127**
  - *kernel regs* ***ar.k0-ar.k7***
  - *RSE control reg* ***ar.rsc***
  - *backing store ptr* ***ar.bsp***
  - *BSP memory ptr* ***ar.bspstore***
  - *RSE NaT collection* ***ar.rnat***
  - *user NaT collection* ***ar.unat***
  - *FP status reg* ***ar.fpsr***
  - *prev frame state* ***ar.pfs***
  - *loop counter* ***ar.lc***
  - *epilog counter* ***ar.ec***
- ***Others***
  - user mask **psr.um**

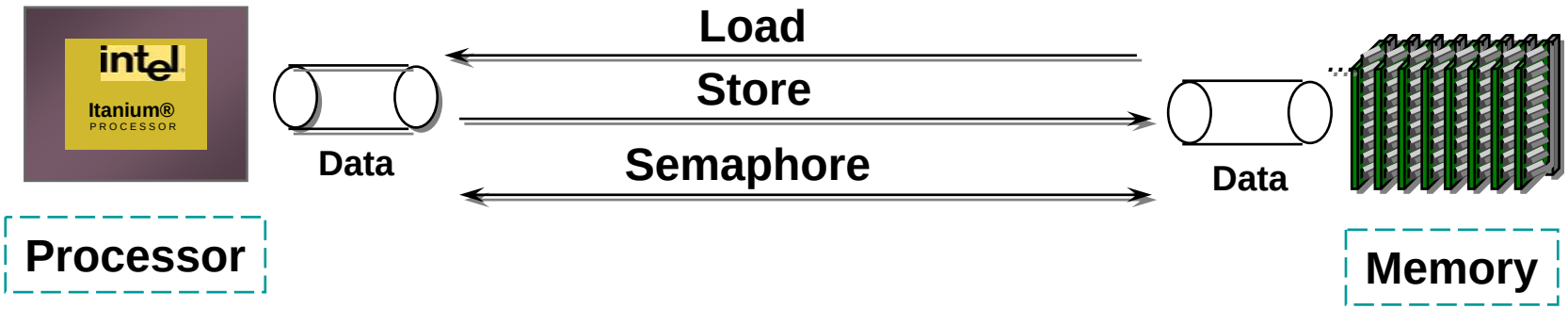
Italics font indicates alias or alternate names.

Compare Instructions Set Predicates

Mnemonic	Operation
cmp/cmp4	GR compare (64-bit/32-bit)
tbit/tnat	Test GR bit/test NaT bit
fcmp	FR compare
fclass	FR class
frcpa/frsqrta	FP reciprocal approx FP reciprocal sq root approx



Memory Access Instructions



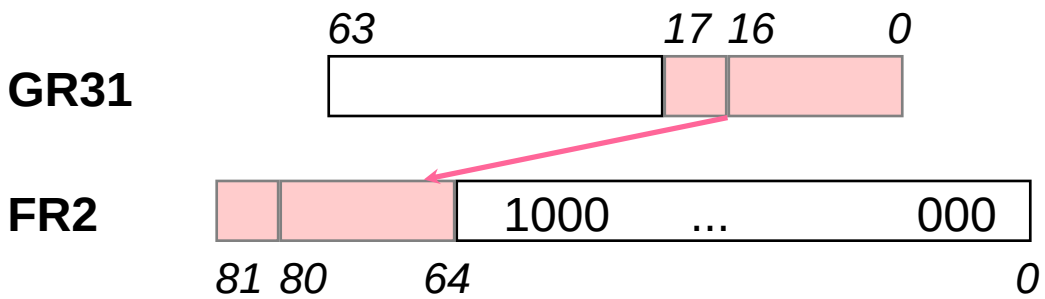
- Memory addressing by byte (64-bit address)
  - It is specified by the contents of a general register.
  - Load/store can also specify base-address register update.
- Byte ordering
  - Big-endian or little-endian data accesses are determined by UM.be bit, a user readable bit in the Processor Status Register.
- Data elements should be stored (aligned) on natural boundaries
  - Hardware fault is on misaligned references.
  - 10-byte floats are stored on 16-byte boundary.
  - Instruction bundles are 16 bytes long and always aligned on 16-byte boundaries.

Floating-Point Data Move

Type	Operation	Mnemonic
FP Memory Access	Load to FR Load pair to FR Store from FR	ldfs, ldf8, ldfd, ldfe, ldf.fill ldfps, ldfp8, ldfpd stfs, stf8, stfd, stfe, stf.spill
FR to/from GR Transfer	GR to FR FR to GR	setf.s, setf.d, setf.exp, setf.sig getf.s, getf.d, getf.exp, getf.sig

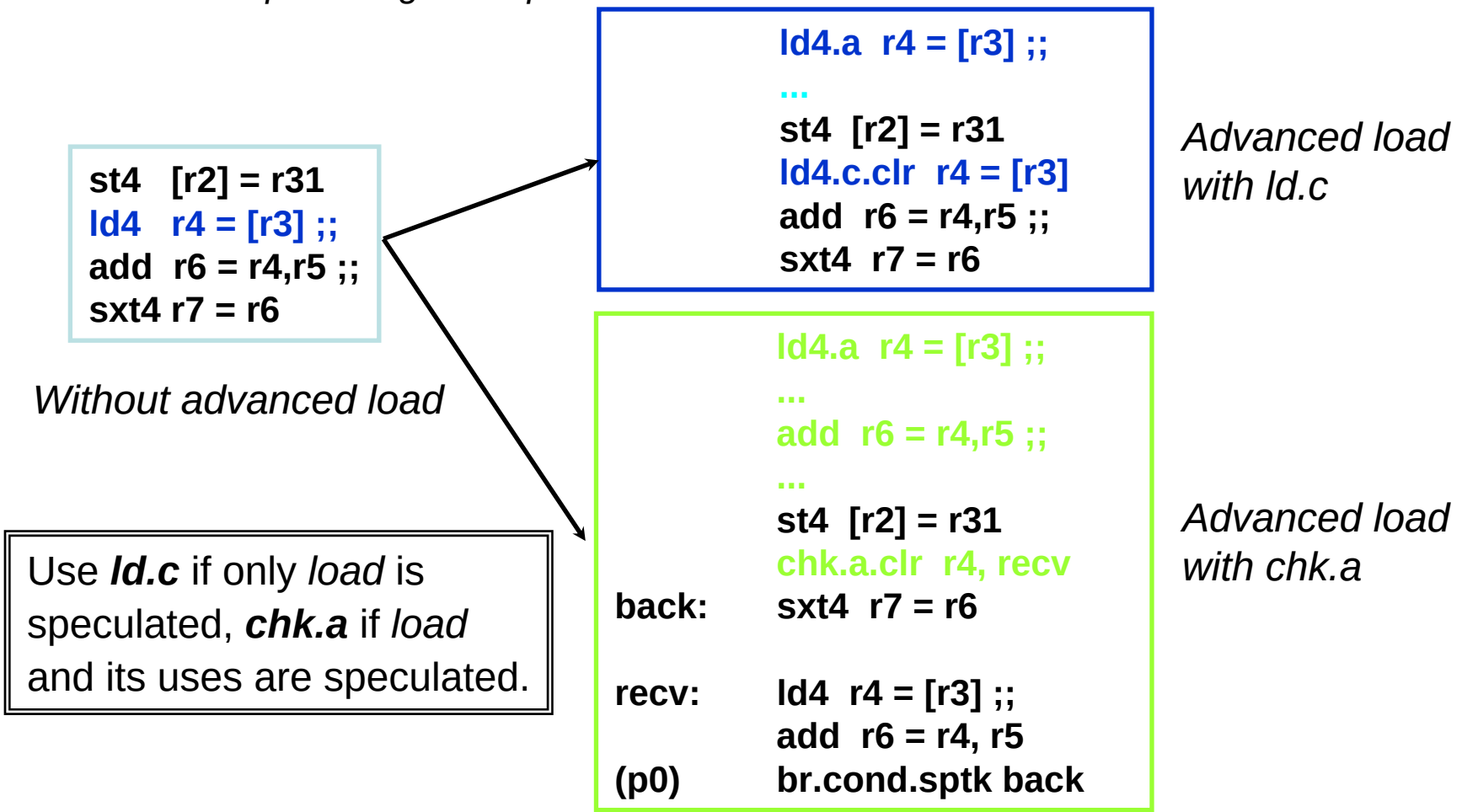
Note: This is a subset of FP instructions.

Transfer GR1 to FR2 exp and sign



Data Speculation

- *Code example: using data speculation*



Software Pipelined Loops

int csum(double a[], double b[], double c[], double x, int max)
for (i=0; i<max;i++)a[i]=b[i]+x*c[i];

Intel® Itanium® processor optimized assembler:

```
.b1_2:
{   .mmi
 (p16)     ldfd        f32=[r34],8
 (p16)     ldfd        f37=[r33],8
           nop.I       0 ;;
} { .mfb
 (p24)     stfd        [r32]=f46,8
 (p20)     fma.d       f42=f8,f36,f41
    br.ctop.sptk       .b1_2 ;;
```

Itanium® -2 processor optimized assembler:

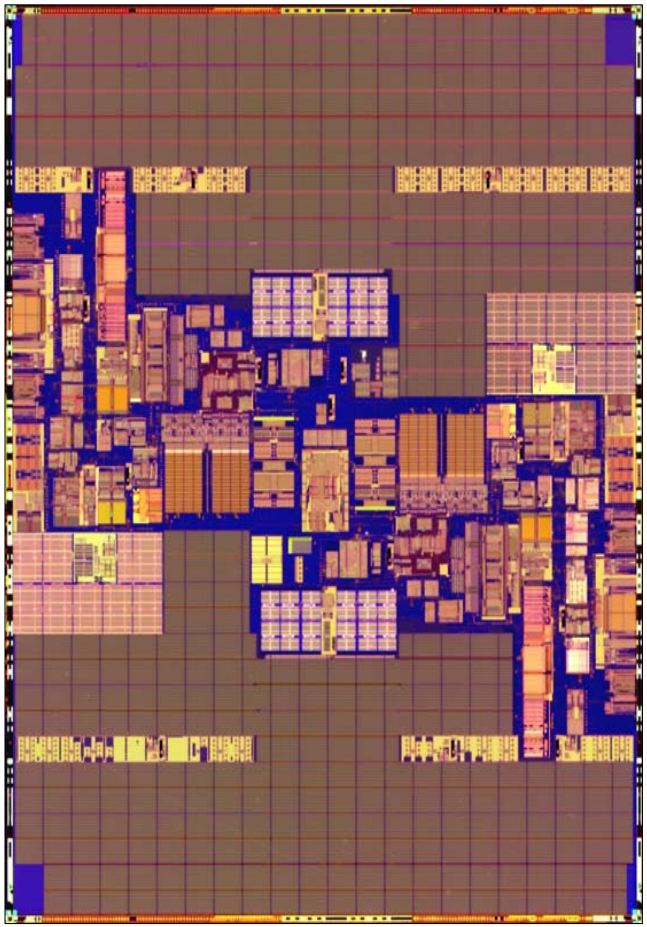
```
.b1_2:
{   .mfi
 (p16)     ldfd        f32=[r34],8
 (p22)     fma.d       f46=f8,f38,f45
           nop.i       0
} { .mmb
 (p16)     ldfd        f39=[r33],8
 (p26)     stfd        [r32]=f50,8
    br.ctop.sptk       .b1_2 ;;
```

Itanium-2® processor can do 4 FP load/store operations/cycle, Itanium® only 2 !!
(double performance for large *max*)

L2 Latency
Itanium: 8
Itanium-2: 6

A very brief Overview on Montecito Processor

Itanium Dual Core: Montecito

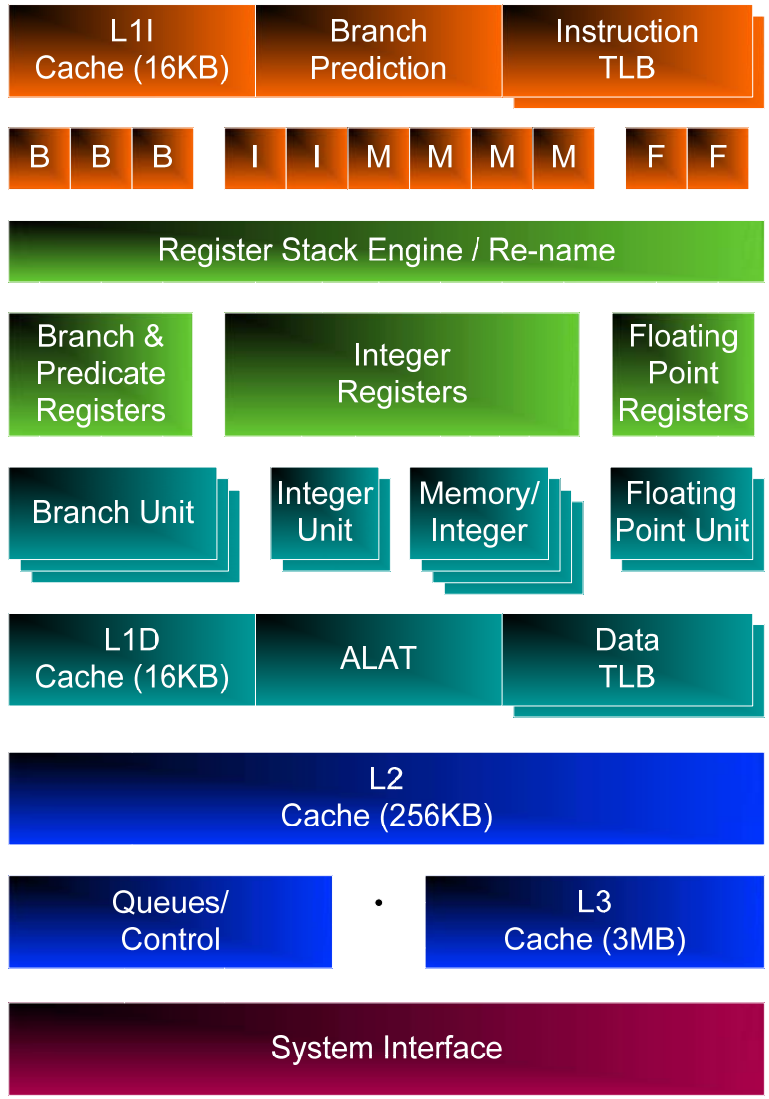


Montecito Feature Summary	
Simultaneous Threads	4
Process technology	90nm
L1 Cache	2 x (D16K + I16K)
L2 Data	2 x 256K
L2 Instruction Cache	2 x 1 MB
L3 Cache (Unified)	2 x 12 MB
Transistors	1,720,000,000
Availability Target	2006

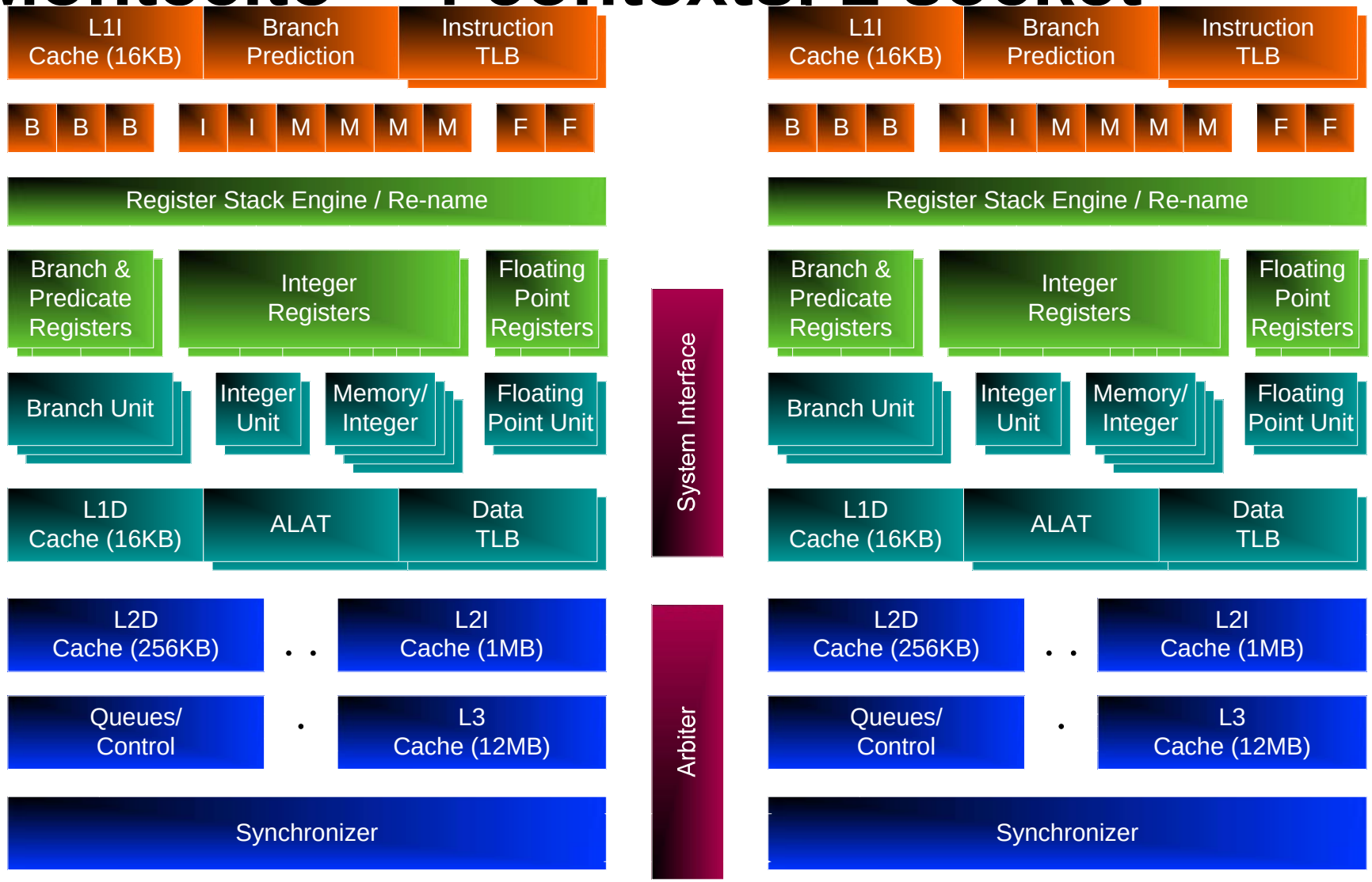
Montecito

Building on Itanium® 2

- Capability
  - Same capable micro-architecture supports familiar optimizations
  - Same system interface allows for leveraged investment
- Efficiency
  - 2 cores, 2 threads, 26.5MByte of cache, and 1.72 billion transistors at 100W
- Extended by new Technologies
  - Foxton
  - Demand Based Switching
  - Pellston
  - Virtualization



Montecito – 4 contexts, 1 socket



Montecito Hyper-Threading

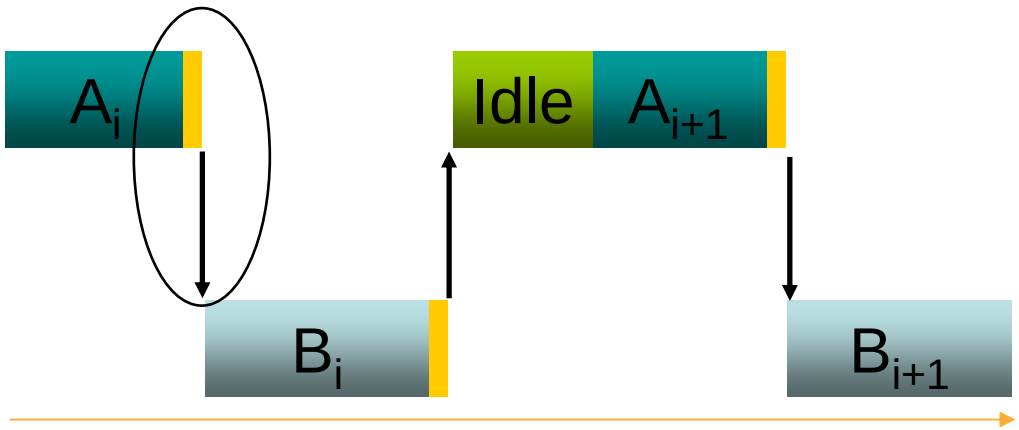
A blend of SMT and TMT

- Temporal multi-threading (TMT) for pipeline
  - Core pipeline never used by two threads simultaneously
  - Different from Hyper-Threading on Intel Pentium® and XEON™ processor which is SMT – simultaneous multi-threading
- Simultaneous multi-threading (SMT) for memory sub-system
  - Memory transfers of both threads can be active at the very same time
- Switching triggered by events
  - Misses on highest-level (L3-) cache
  - Expiration of timer
  - Hysteresis to avoid needless switches
  - `hint@pause` gives software control
    - Typically done as part of spin-wait-loop

Montecito Hyper-/Multi-threading

Serial Execution

Montecito Multi-threaded Execution



Multi-threading decreases stalls and increase performance

Montecito Core Extensions

Slightly extended Itanium® 2 processor core:

- Larger atomic ops: 16-byte ld16/st16/cmp8xchg16
  - Support non-blocking synchronization in database apps
  - Improves performance scalability of database applications on large SMP
- Instruction(s) to support virtualization
- Cash flush extensions (fc and fc.i)
- hint@pause for thread switching
  - Is a NOP on older architecture, will not fault
- Additional integer shifter
  - Allows scheduling two variable shifts per cycle
  - Enhanced processing performance in cryptographic codes
- Faster chk.a/chk.s resteer
- Support in compiler 9.0 via
  - intrinsics (not documented yet)
  - Introduction of new machine model (KNOBs file for Montecito)

Montecito & SW Development

- Itanium® 2 Architecture compatible in general
  - Any Itanium® code will work fine
  - Use of new instructions might raise illegal op-code exception on non-Montecito processors
  - Developer- and compiler relevant key features of caches will not change (transparent L2-I cache and larger L3 cache)
  - For some application (cryptographic codes) the new execution units make a difference
- Cycle counting needs more attention due to Demand Based Switching and Foxton Technology
  - Hardware provide 2 “cycle” counters for fixed normalized frequency and exact cycle counting
- Threading will make a difference of course

Performance Monitoring

Feature	Itanium2	Montecito
Event Counter Regs	4	12 (per thread)
Branch Trace Buffer	8	16 (per threadd)
Lev 2 Cache	restricted	Less restricted
Opcode filtering	partial	Full 41 bits
IP trace capture	No	yes
New events		L2 I-cache, L3 state, Multi Threading, Foxton, Arbiter

Summary

- Itanium® architecture offers many sophisticated features unavailable in other architectures like an increase in register count by a magnitude
- Software pipelining is key to get best performance for HPC applications
- EPIC architecture requires more compiler and developer help to achieve optimal machine instruction selection and scheduling
  - No “out-of-order” help as for classical architecture (Pentium® 4, XEON™)
- Montecito will introduce hardware-supported multi-threading, both by multiple cores and hyper-threading

Questions?